good luck :)

UNIT 1

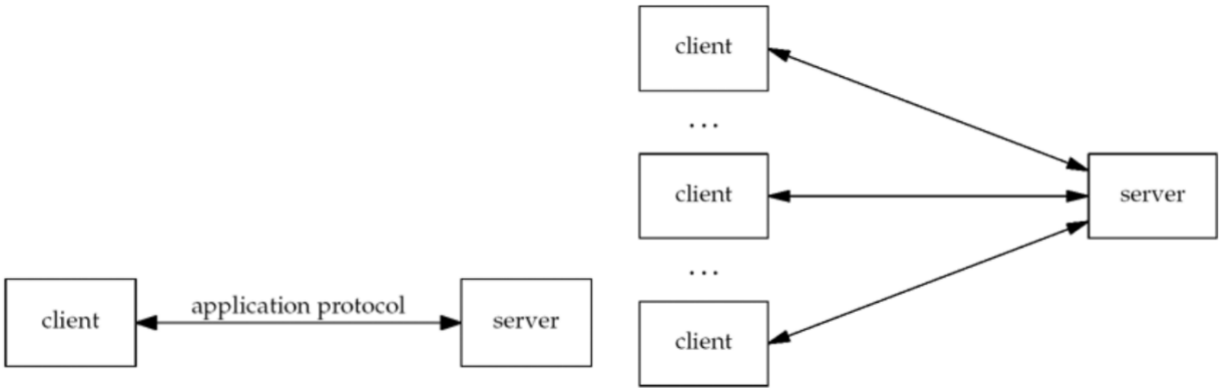
1. **What is network protocol? With a neat block diagram explain the network application for client and server.**

Ans:

A network protocol is an established set of rules that determine how data is transmitted between different devices in the same network.

Essentially, it allows connected devices to communicate with each other, regardless of any differences in their internal processes, structure or design.

Network applications: client and server



- Clients normally communicate with one server at a time, although using a Web browser as an example, we might communicate with many different Web servers over, say, a 10-minute time period.
- From the server's perspective, at any given point in time, it is not unusual for a server to be communicating with multiple clients. We show this in Figure.
- The client application and the server application may be thought of as communicating via a network protocol, but actually, multiple layers of network protocols are typically involved.
- TCP/IP protocol suite, also called the Internet protocol suite. For example, Web clients and servers communicate using the Transmission Control Protocol, or TCP.
- TCP, in turn, uses the Internet Protocol, or IP, and IP communicates with a data link layer of some form.

good luck :)

2. **List out the various approaches used to handle multiple clients at the same time.**

Ans:

Multi threading :

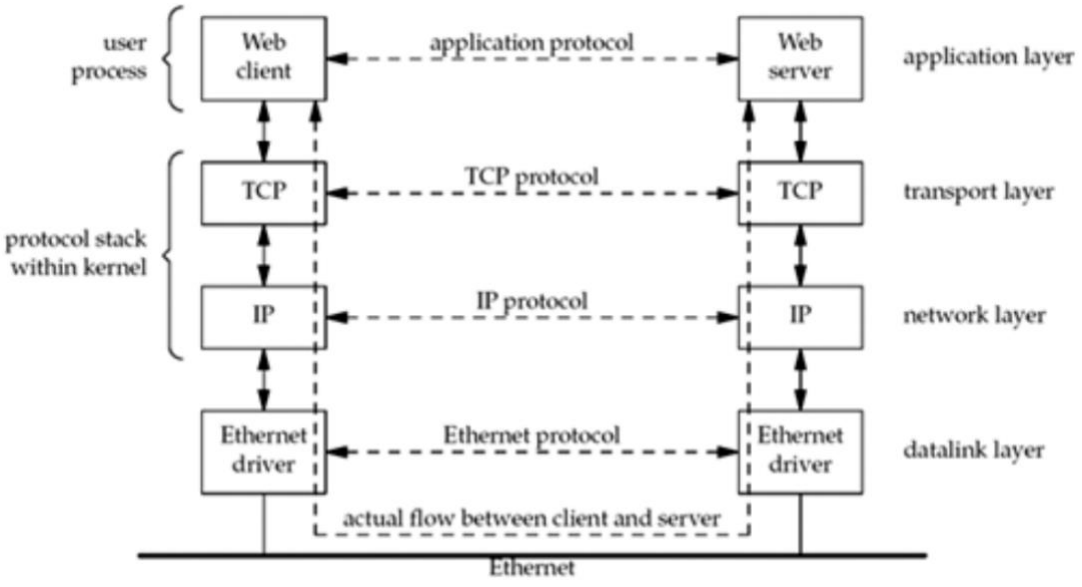
- The simple way to handle multiple clients would be to spawn a new thread for every new client connected to the server.
- Semaphore is simply a variable that is non-negative and shared between threads. This variable is used to solve the critical section problem and to achieve process synchronization in the multiprocessing environment.

Socket Programming:

- Above method is strongly not recommended because of various disadvantages, namely:
    - Threads are difficult to code, debug and sometimes they have unpredictable results.
    - Overhead switching of context
    - Not scalable for large number of clients
    - Deadlocks can occur
- When you design a server program, plan for multiple concurrent processes. Special socket calls are available for that purpose they are called concurrent servers, as opposed to the more simple type of iterative server.

3. **With a neat block diagram, explain the client and server communication on Local Area Network using TCP.**

Ans:



- Even though the client and server communicate using an application protocol, the transport layers communicate using TCP.
- The actual flow of information between client and server goes down the protocol stack on one side, across the network, and up the protocol stack on the other side.

- Client & Server are typically user processes, while TCP and IP protocols are normally part of the protocol stack within the kernel.
- The four layers labeled in the diagram are the Application layer, Transport layer, Network layer, Data-link layer.
- Some clients and servers use the User Datagram Protocol (UDP) instead of TCP.

4. List and explain the steps involved in a simple daytime client.

Ans:

1. **Include headers:** headers includes numerous system headers that are needed by most network programs and defines various constants that we use .
2. **Command-line arguments:** definition of the main function along with the command-line arguments.
3. **Create TCP socket :** The socket function creates an Internet (AF_INET) stream (SOCK_STREAM) socket, which is a fancy name for a TCP socket. The function returns a small integer descriptor that we can use to identify the socket in all future function calls
4. **Specify server's IP address and port :** Fill in an Internet socket address structure (sockaddr_in) with the server's IP address and port number. We set the entire structure to 0 using bzero, set the address family to AF_INET, set the port number to 13, and set the IP address to the value specified as the first command-line argument (argv[1]).
5. **Establish connection with server:** The connect function, when applied to a TCP socket, establishes a TCP connection with the server specified by the socket address structure.
6. **Read and display server's reply:** We read the server's reply and display the result using the standard I/O functions.
7. **Terminate program:** exit terminates the program. Unix always closes all open descriptors when a process terminates, so our TCP socket is now closed.

good luck :)

5. Develop the 'C' program to implement a simple daytime client.

Ans:

```c
//TCP Day Time Client

#include "unp.h"

int main(int argc, char **argv) {
    if (argc != 2)
        err_quit("usage: a.out <IPaddress>");

    char receive[30];

    int sockfd = Socket(AF_INET, SOCK_STREAM, 0);

    struct sockaddr_in serverAddress;
    bzero(&serverAddress, sizeof(serverAddress));
    serverAddress.sin_family = AF_INET;
    Inet_pton(AF_INET, argv[1], &serverAddress.sin_addr);
    serverAddress.sin_port = htons(13);

    Connect(sockfd, (SA *)&serverAddress, sizeof(serverAddress));

    Recv(sockfd, receive, 29, 0);
    receive[30] = '\0';

    printf("%s", receive);
}
```

6. Comment on the Protocol Independence. Modify the day time client program for IPv6.

Ans:

It is better to make a program protocol-independent. Protocol-independent is achieved by using the **`getaddrinfo`** function (which is called by **`tcp_connect`**). Another deficiency in our programs is that the user must enter the server's IP address as a dotteddecimal number. Humans work better with names instead of numbers.

//Simple Daytime Client program for IPv6
//Refer ans 5 and add 6 wherever necessary.

good luck :)

7. What are wrapper functions? Develop the wrapper function for the following:

a. Socket function

b. Pthread_mutex_lock

Ans:

A wrapper function is a subroutine in a software library or a computer program whose main purpose is to call a second subroutine or a system call with little or no additional computation.

a. Socket function:

```
int Socket(int family, int type, int protocol)
{
      int n;
      if ( (n = socket(family, type, protocol)) < 0)
            err_sys("socket error");
      return (n);
}
```

b. Pthread_mutex_lock:

```
Void Pthread_mutex_lock(pthread_mutex_t *mptr)
{
      int n;
      if ( (n = pthread_mutex_lock(mptr)) == 0)
            return;
      errno = n;
      err_sys("pthread_mutex_lock error");
}
```

8. List and explain the steps involved in a simple daytime server.

Ans:

1. Create a TCP Socket:
    a. The creation of the TCP socket is identical to the client code.
2. Bind server's well-known port to socket:
    a. The server's well-known port (13 for the daytime service) is bound to the socket by filling in an Internet socket address structure and calling **`bind`**.
    b. We specify the IP address as **`INADDR_ANY`**, which allows the server to accept a client connection on any interface, in case the server host has multiple interfaces.
3. Convert socket to listening socket:
    a. By calling **`listen`**, the socket is converted into a listening socket, on which incoming connections from clients will be accepted by the kernel.
    b. These three steps, **`socket`**, **`bind`**, and **`listen`**, are the normal steps for any TCP server to prepare what we call the listening descriptor (**`listenfd`** in this example).
    c. The constant **`LISTENQ`** is from our **`unp.h`** header. It specifies the maximum number of client connections that the kernel will queue for this listening descriptor.

4. Accept client connection, send reply:
    a. The server process is put to sleep in the call to **`accept`**, waiting for a client connection to arrive and be accepted.
    b. A TCP connection uses what is called a three-way handshake to establish a connection.
    c. When this handshake completes, **`accept`** returns, and the return value from the function is a new descriptor (**`connfd`**) that is called the connected descriptor.
    d. This new descriptor is used for communication with the new client.
    e. A new descriptor is returned by **`accept`** for each client that connects to our server.
    f. The current time and date are returned by the library function **`time`**, which returns the number of seconds since the Unix Epoch: **`00:00:00`** January 1, 1970, Coordinated Universal Time (UTC).
    g. The next library function, **`ctime`**, converts this integer value into a human-readable string such as **`Mon May 26 20:58:40 2003`**
    h. A carriage return and linefeed are appended to the string by snprintf, and the result is written to the client by **`write`**.
5. Terminate connection:
    a. The server closes its connection with the client by calling close.
    b. This initiates the normal TCP connection termination sequence: a FIN is sent in each direction and each FIN is acknowledged by the other end.
    c. Program is protocol dependent.
    d. Server handles only one client at a time.
    e. If multiple client connections arrive at about the same time, the kernel queues them, up to some limit, and returns them to **`accept`** one at a time.

good luck :)

9. Develop the 'C' program to implement a simple daytime server.

Ans:

```c
//TCP Day Time Server

#include <time.h>
#include "unp.h"

int main(int argc, char **argv) {
    int listenfd = Socket(AF_INET, SOCK_STREAM, 0);

    struct sockaddr_in serverAddress;
    bzero(&serverAddress, sizeof(serverAddress));
    serverAddress.sin_family = AF_INET;
    serverAddress.sin_addr.s_addr = INADDR_ANY;
    serverAddress.sin_port = htons(13);

    Bind(listenfd, (SA *)&serverAddress, sizeof(serverAddress));

    Listen(listenfd, LISTENQ);

    for (;;) {
        int client = Accept(listenfd, NULL, NULL);
        time_t ticks = time(NULL);
        Send(client, ctime(&ticks), 30, 0);
        Close(client);
    }
}
```

10. Write a note on Unix errno value.

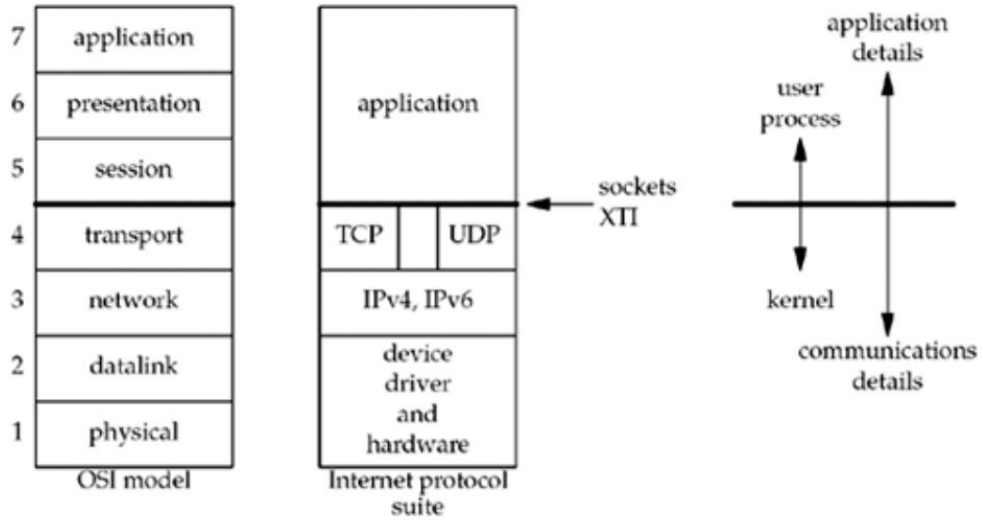
Ans:

- When an error occurs in a Unix function (such as one of the socket functions), the global variable errno is set to a positive value indicating the type of error
- **`err_sys`** function looks at the value of errno and prints the corresponding error message string (e.g., "Connection timed out" if errno equals **`ETIMEDOUT`**)
- The value of errno is set by a function only if an error occurs.
- Its value is undefined if the function does not return an error.
- All of the positive error values are constants with all-uppercase names beginning with "E," and are normally defined in the **`<sys/errno.h>`** header.

good luck :)

11. Explain with a neat block diagram the layers of OSI model and Internet protocol suite.

Ans:



- A common way to describe the layers in a network is to use the International Organization for Standardization (ISO) open systems interconnection (OSI) model for computer communications. This is a seven-layer model,
- Datalink+ physical = drivers and network hardware.
- The network layer is handled by the IPv4 and IPv6 protocols
- We show a gap between TCP and UDP to indicate that it is possible for an application to bypass the transport layer and use IPv4 or IPv6 directly. This is called a raw socket.
- The upper three layers of the OSI model are combined into a single layer called the application. This is the Web client (browser), Telnet client, Web server, FTP server, or whatever application we are using.

12. List and explain the features of UDP Protocol in detail.

Ans:

- UDP is a simple transport-layer protocol.
- The application writes a message to a UDP socket, which is then encapsulated in a UDP datagram, which is then further encapsulated as an IP datagram, which is then sent to its destination.
- There is no guarantee that a UDP datagram will ever reach its final destination, that order will be preserved across the network, or that datagrams arrive only once.
- If a datagram reaches its final destination but the checksum detects an error, or if the datagram is dropped in the network, it is not delivered to the UDP socket and is not automatically retransmitted.
- To ensure reliability, we can build lots of features into our application: acknowledgments from the other end, timeouts, retransmissions, and so on.
- Each UDP datagram has a length. The length of a datagram is passed to the receiving application along with the data.

- UDP provides a connectionless service, as there need not be any long-term relationship between a UDP client and server.
- For example, a UDP client can create a socket and send a datagram to a given server and then immediately send another datagram on the same socket to a different server.
- Similarly, a UDP server can receive several datagrams on a single UDP socket, each from a different client.

13. List and explain the features of TCP Protocol in detail.

Ans:

TCP (Transmission Control Protocol) is a transport layer protocol in the TCP/IP protocol suite. It is responsible for establishing and maintaining a reliable, error-free communication channel between two devices on a network. The main features of TCP include:

//ChatGPT

1. Connection-oriented: TCP uses a three-way handshake to establish a reliable, point-to-point connection between two devices before data can be exchanged. This ensures that the data is received by the intended recipient and that any errors are detected and corrected.
2. Flow control: TCP uses a sliding window mechanism to regulate the flow of data between the sender and receiver. This prevents the sender from overwhelming the receiver with too much data at once.
3. Error detection and correction: TCP includes error detection and correction mechanisms to ensure that data is transmitted correctly and that any errors are detected and corrected. This includes checksums, retransmission, and flow control.
4. Congestion control: TCP uses a congestion control algorithm to prevent network congestion and ensure that data is transmitted efficiently. This includes algorithms such as slow start, congestion avoidance, and fast retransmit.
5. Sequencing: TCP assigns a sequence number to each segment of data that is transmitted, allowing the receiver to reassemble the data in the correct order.
6. Acknowledgments: TCP uses acknowledgments to confirm that data has been received successfully and to signal the sender to retransmit any lost data.
7. Port numbers: TCP uses port numbers to identify different applications and services running on a device.
8. Urgent data: TCP can also support urgent data, which is marked with a special flag and received ahead of other data in the buffer.
9. Graceful close: TCP uses a graceful close mechanism to ensure that all data is transmitted before a connection is closed.
10. Stateful protocol: TCP is a stateful protocol as it keeps track of the state of the connection.

-OR-

good luck :)

//Textbook

- TCP provides connections between clients and servers.
- A TCP client establishes a connection with a given server, exchanges data with that server across the connection, and then terminates the connection.
- **Reliability:** TCP also provides reliability. When TCP sends data to the other end, it requires an acknowledgment in return. If an acknowledgment is not received, TCP automatically retransmits the data and waits a longer amount of time. After some number of retransmissions, TCP will give up, with the total amount of time spent trying to send data typically between 4 and 10 minutes.
- TCP **does not guarantee that the data will be received by the other endpoint**: as this is impossible. It delivers data to the other endpoint if possible, and notifies the user if it is not possible.
- **Estimates RTT:** TCP uses algorithm to estimate round-trip time (RTT) between a client and server dynamically so that it knows how long to wait for an acknowledgment.
    - For example, the RTT on a LAN can be milliseconds while across a WAN, it can be seconds.
    - TCP continuously estimates the RTT of a given connection, because the RTT is affected by variations in the network traffic.
- **Data Sequencing:** Sequences the data by associating a sequence number with every byte that it sends.
    - For example, assume an application writes 2,048 bytes to a TCP socket, causing TCP to send two segments, the first containing the data with sequence numbers 1–1,024 and the second containing the data with sequence numbers 1,025–2,048.
    - If the segments arrive out of order, the receiving TCP will reorder the two segments based on their sequence numbers before passing the data to the receiving application.
    - If TCP receives duplicate data from its peer, it can detect that the data has been duplicated, and discard the duplicate data.
- **Flow Control:**
    - For example, the RTT on a LAN can be milliseconds while across a WAN, it can be seconds. Furthermore, TCP continuously estimates the RTT of a given connection, because the RTT is affected by variations in the network traffic.
    - At any time, the window is the amount of room currently available in the receive buffer, guaranteeing that the sender cannot overflow the receive buffer.
    - The window changes dynamically over time.
- **Full-Duplex:**
    - An application can send and receive data in both directions on a given connection at any time.
    - TCP must keep track of state information such as sequence numbers and window sizes for each direction of data flow.

good luck :)

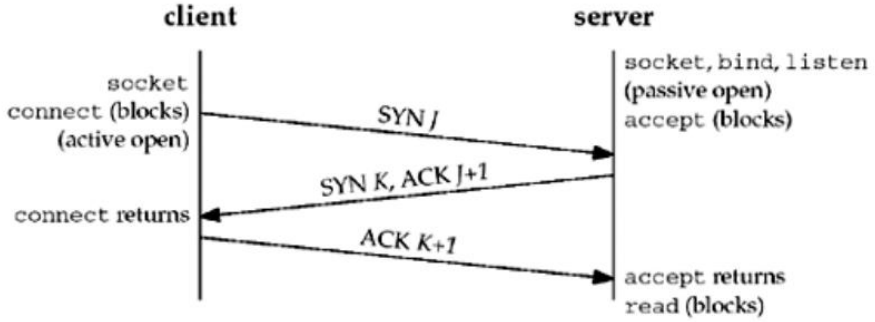
14. Explain with a neat diagrams the following:

a. **TCP connection establishment**
b. **TCP data transfer**
c. **TCP connection termination**

Ans:

a. TCP connection establishment

- The server must be prepared to accept an incoming connection. This is normally done by calling socket, bind, and listen and is called a passive open.
- The client issues an active open by calling connect. This causes the client TCP to send a "synchronize" (SYN) segment, which tells the server the client's initial sequence number for the data that the client will send on the connection. Normally, there is no data sent with the SYN; it just contains an IP header, a TCP header, and possible TCP options (which we will talk about shortly).
- The server must acknowledge (ACK) the client's SYN and the server must also send its own SYN containing the initial sequence number for the data that the server will send on the connection. The server sends its SYN and the ACK of the client's SYN in a single segment.
- The client must acknowledge the server's SYN.
- The minimum number of packets required for this exchange is three; hence, this is called TCP's three-way handshake.



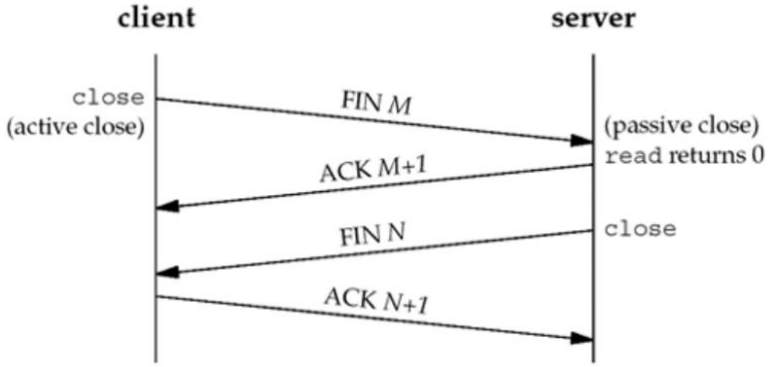
b. TCP data transfer

//Add if anyone gets.

c. TCP connection termination

- One application calls close first, and we say that this end performs the active close. This end's TCP sends a FIN segment, which means it is finished sending data.
- The other end that receives the FIN performs the passive close. The received FIN is acknowledged by TCP. The receipt of the FIN is also passed to the application as an end-of-file (after any data that may have already been queued for the application to receive), since the receipt of the FIN means the application will not receive any additional data on the connection.
- Sometime later, the application that received the end-of-file will close its socket. This causes its TCP to send a FIN.

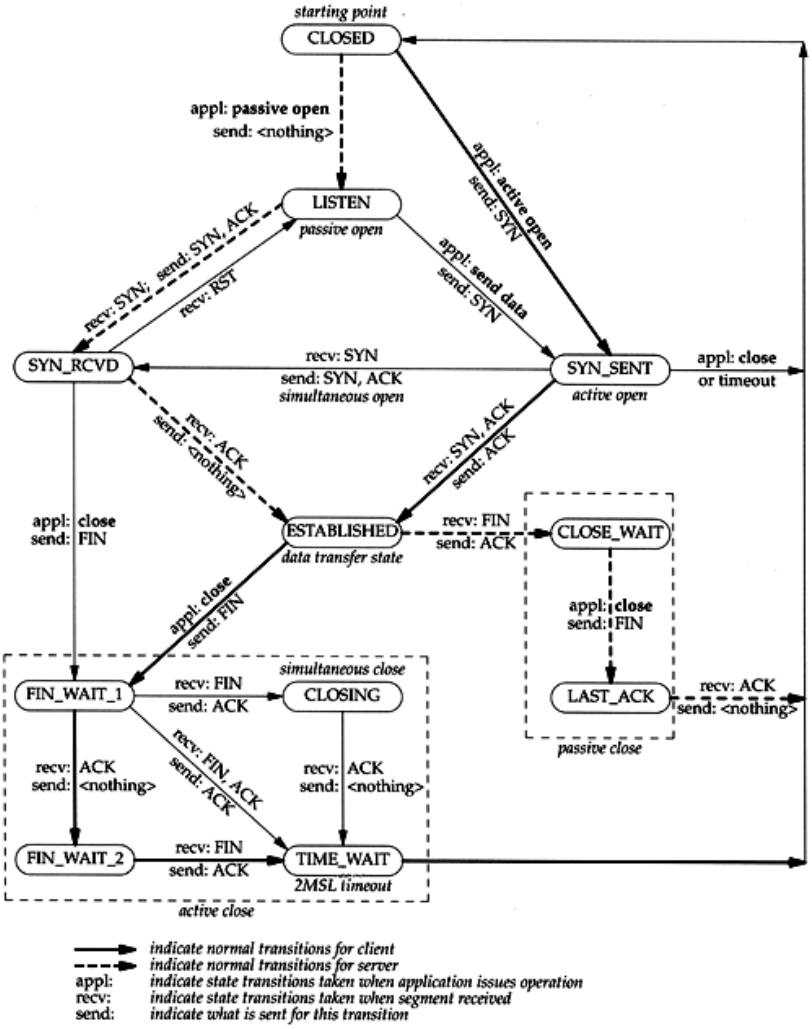
- The TCP on the system that receives this final FIN (the end that did the active close) acknowledges the FIN.



15. Explain the TCP State Transition diagram.

good luck :)

Ans:



- There are 11 different states defined for a connection and the rules of TCP dictate the transitions from one state to another, based on the current state and the segment received in that state.
    - For example, if an application performs an active open in the CLOSED state, TCP sends a SYN and the new state is SYN_SENT.
    - If TCP next receives a SYN with an ACK, it sends an ACK and the new state is ESTABLISHED.
    - This final state is where most data transfer occurs.
    - The two arrows leading from the ESTABLISHED state deal with the termination of a connection.
    - If an application calls close before receiving a FIN (an active close), the transition is to the FIN_WAIT_1 state.
    - But if an application receives a FIN while in the ESTABLISHED state (a passive close), the transition is to the CLOSE_WAIT state.
- We denote the normal client transitions with a darker solid line and the normal server transitions with a darker dashed line.

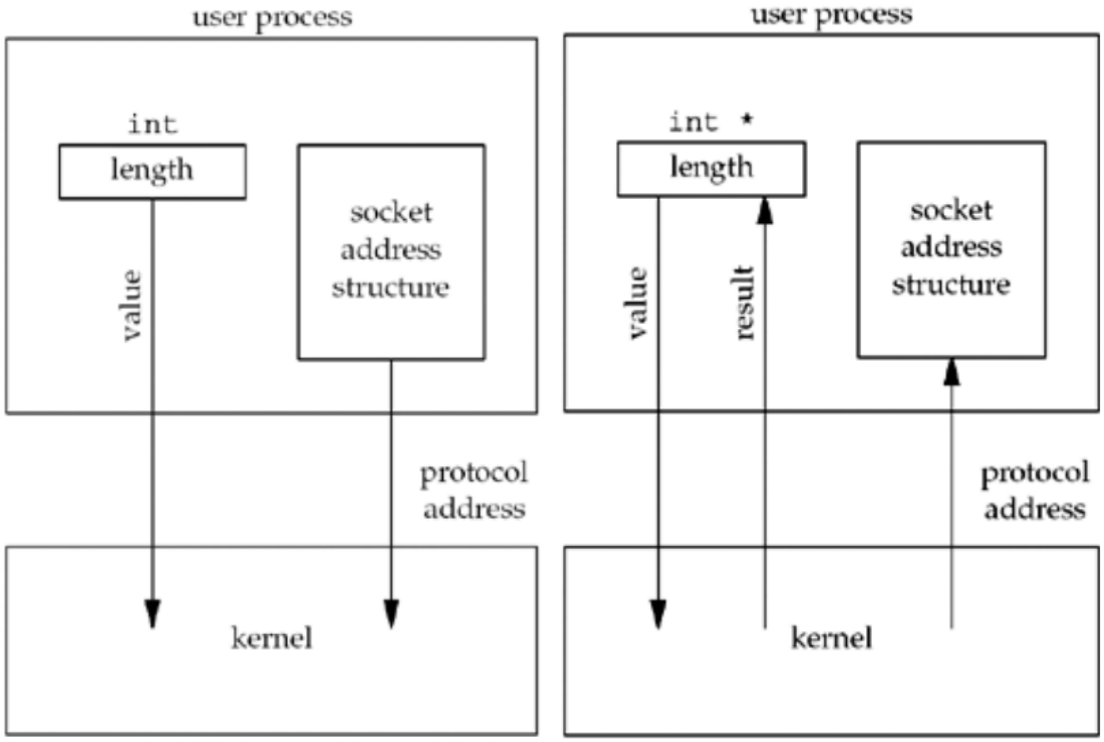
good luck :)

Unit 2: (Sockets Introduction)

1. **What are Value- Result Arguments? Explain the scenario with a neat block diagram.**

Ans:

- When a socket address structure is passed to any socket function, it is always passed by reference. That is, a pointer to the structure is passed. The length of the structure is also passed as an argument. But the way in which the length is passed depends on which direction the structure is being passed: from the process to the kernel, or vice versa.
- **`bind`**, **`connect`**, and **`sendto`**, pass a socket address structure from the process to the kernel.

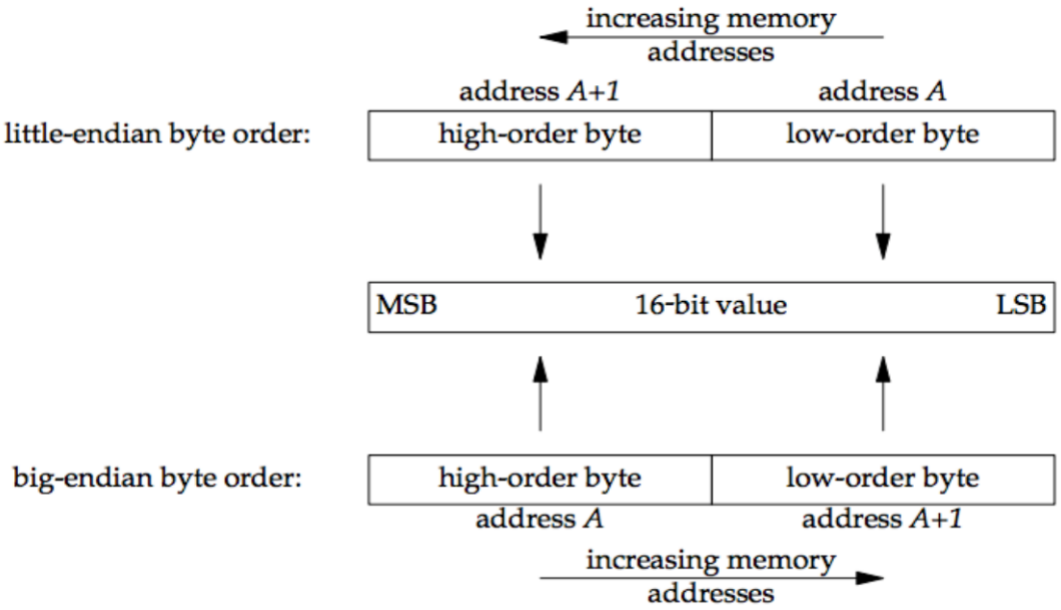


- Kernel is passed both the pointer and the size of what the pointer points to, it knows exactly how much data to copy from the process into the kernel.
- **`accept`**, **`recvfrom`**, **`getsockname`**, and **`getpeername`**, pass a socket address structure from the kernel to the process.
- **`connect (sockfd, (SA *) &serv, sizeof(serv));`**
- Size changes from an integer to be a pointer to an integer is because the size is both a value when the function is called and a result when the function returns.
- **`getpeername(unixfd, (SA *) &cli, &len);`**
- Two other functions pass socket address structures: **`recvmsg`** and **`sendmsg`**.
- When using value-result arguments for the length of socket address structures, if the socket address structure is fixed-length, the value returned by the kernel will always be that fixed size: 16 for an IPv4 **`sockaddr_in`** and 28 for an IPv6 **`sockaddr_in6`**.

good luck :)

2. Explain with a neat diagram the various byte ordering functions.

Ans:



- For a 16-bit integer that is made up of 2 bytes, there are two ways to store the two bytes in memory:
  - **Little-endian order:** low-order byte is at the starting address.
  - **Big-endian order:** high-order byte is at the starting address.
- In this figure, we show increasing memory addresses going from right to left in the top, and from left to right in the bottom.
- We also show the most significant bit (MSB) as the leftmost bit of the 16-bit value and the least significant bit (LSB) as the rightmost bit.
- The terms "little-endian" and "big-endian" indicate which end of the multibyte value, the little end or the big end, is stored at the starting address of the value.
- We store the two-byte value 0x0102 in the short integer and then look at the two consecutive bytes, c[0] (the address A) and c[1] (the address A+1) to determine the byte order.
- The string **`CPU_VENDOR_OS`** is determined by the GNU autoconf program.
  - Networking protocols must specify a network byte order. The sending protocol stack and the receiving protocol stack must agree on the order in which the bytes of these multibyte fields will be transmitted. The Internet protocols use big-endian byte ordering for these multibyte integers.
  - But, both history and the POSIX specification say that certain fields in the socket address structures must be maintained in network byte order. We use the following four functions to convert between these two byte orders:

good luck :)

```
#include <netinet/in.h>
uint16_t htons(uint16_t host16bitvalue);
uint32_t htonl(uint32_t host32bitvalue);
/* Both return: value in network byte order */
uint16_t ntohs(uint16_t net16bitvalue);
uint32_t ntohl(uint32_t net32bitvalue); /* Both return: value in host byte order */
```

- h stands for *host*
- n stands for *network*
- s stands for *short* (16-bit value, e.g. TCP or UDP port number)
- l stands for *long* (32-bit value, e.g. IPv4 address)

3. Write a note on the Byte Manipulation Functions.

Ans:

- There are two groups of functions that operate on multibyte fields, without interpreting the data, and without assuming that the data is a null-terminated C string. We need these types of functions when dealing with socket address structures because we need to manipulate fields such as IP addresses, which can contain bytes of 0, but are not C character strings. The functions beginning with **`str`** (for string), defined by including the **`<string.h>`** header, deal with null-terminated C character strings.
- The first group of functions, whose names begin with **`b`** (for byte), are from 4.2BSD and are still provided by almost any system that supports the socket functions. The second group of functions, whose names begin with **`mem`** (for memory), are from the ANSI C standard and are provided with any system that supports an ANSI C library.
- We first show the Berkeley-derived functions, although the only one we use in this text is **`bzero`**. You may encounter the other two functions, **`bcopy`** and **`bcmp`**, in existing applications.

```
#include <string.h>
void bzero(void *dest, size_t nbytes);
void bcopy(const void *src, void *dest, size_t nbytes);
int bcmp(const void *ptr1, const void *ptr2, size_t nbytes);
/* Returns: 0 if equal, nonzero if unequal */
```

- **`bzero`** sets the specified number of bytes to 0 in the destination. We often use this function to initialize a socket address structure to 0.
- **`bcopy`** moves the specified number of bytes from the source to the destination.
- **`bcmp`** compares two arbitrary byte strings. The return value is zero if the two byte strings are identical; otherwise, it is nonzero.

```
void *memset(void *dest, int c, size_t len);
void *memcpy(void *dest, const void *src, size_t nbytes);
int memcmp(const void *ptr1, const void *ptr2, size_t nbytes);
/* Returns: 0 if equal, <0 or >0 if unequal (see text) */
```

good luck :)

- **memset** sets the specified number of bytes to the value c in the destination.
- **memcpy** is similar to bcopy, but the order of the two pointer arguments is swapped.
- **memcmp** compares two arbitrary byte strings and returns 0 if they are identical.

4. Develop a 'C' program to determine host byte order.

Ans:

```c
//Determine Host Byte Order

#include "unp.h"

int main() {
    uint32_t num = 0x01234567;
    uint8_t *n = (uint8_t *) &num;

    if (*n == 0x67)
        printf("Little Endian\n");
    else if (*n == 0x01)
        printf("Big Endian\n");
    else
        printf("Unknown byte order\n");

    return 0;
}
```

5. Illustrate the significance of socket functions for elementary TCP client/server with a neat block diagram.

good luck :)

Ans:

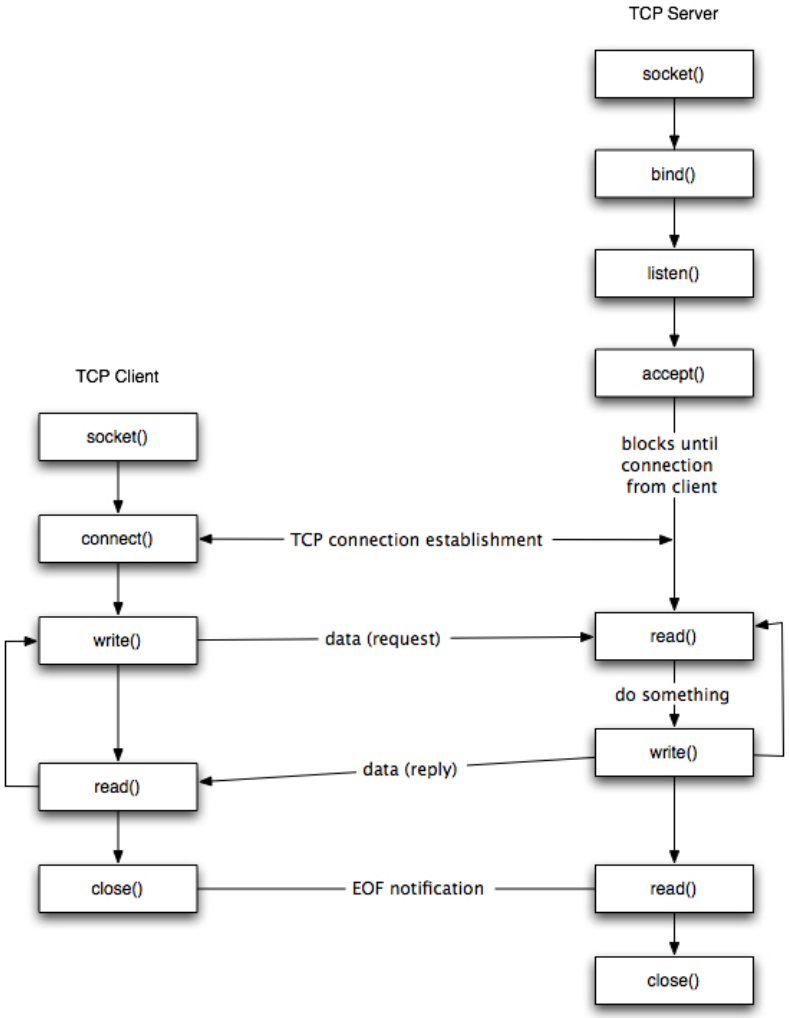


Figure shows a timeline of the typical scenario that takes place between a TCP client and server. First, the server is started, then sometime later, a client is started that connects to the server. We assume that the client sends a request to the server, the server processes the request, and the server sends a reply back to the client. This continues until the client closes its end of the connection, which sends an end-of-file notification to the server. The server then closes its end of the connection and either terminates or waits for a new client connection.

6. **Explain the following arguments of the socket function:**
    a. **Family**

b. **Type**
c. **Protocol**

Ans:

`socket` function:

```
#include<sys/socket.h>
int socket (int family, int type, int protocol);
```

a. **Family:**
family specifies the protocol family and is one of the constants shown in Figure. This argument is often referred to as domain instead of family.

Family	Description
AF_INET	IPv4 protocols
AF_INET6	IPv6 protocols
AF_LOCAL	Unix domain protocols
AF_ROUTE	Routing Sockets
AF_KEY	Ket socket

b. **Type:**
The socket type is one of the constants shown in Figure.

Type	Description
SOCK_STREAM	Stream socket
SOCK_DGRAM	Datagram socket
SOCK_SEQPACKET	Sequenced packet socket
SOCK_RAW	Raw socket

c. **Protocol:**
The protocol argument to the socket function should be set to the specific protocol type found in Figure, or 0 to select the system's default for the given combination of family and type.

Protocol	Description
IPPROTO_TCP	TCP transport protocol
IPPROTO_UDP	UDP transport protocol
IPPROTO_SCTP	SCTP transport protocol

7. **Explain the following functions of TCP socket:**
connect, bind, listen, accept,and close.

Ans:

good luck :)

a. **`connect()`**: This function is used by a TCP client to establish a connection with a server. It takes three arguments: the socket descriptor, a pointer to the server's address (IP and port), and the size of the address structure. The function returns 0 on success, and -1 on failure.

```
#include <sys/socket.h>

int connect(int sockfd, const struct sockaddr *servaddr, socklen_t addrlen);
```

Returns: 0 if OK, -1 on error

b. **`bind()`**: This function is used by a TCP server to bind a socket to a specific address and port. It takes three arguments: the socket descriptor, a pointer to the server's address (IP and port), and the size of the address structure. The function returns 0 on success, and -1 on failure.

```
#include <sys/socket.h>

int bind (int sockfd, const struct sockaddr *myaddr, socklen_t addrlen);
```

Returns: 0 if OK,-1 on error

c. **`listen()`**: This function is used by a TCP server to listen for incoming connections. It takes two arguments: the socket descriptor, and the maximum number of connections that can be queued. The function returns 0 on success, and -1 on failure.

```
#include <sys/socket.h>

#int listen (int sockfd, int backlog);
```

Returns: 0 if OK, -1 on error

d. **`accept()`**: This function is used by a TCP server to accept an incoming connection. It takes three arguments: the socket descriptor, a pointer to the client's address (IP and port), and the size of the address structure. The function returns a new socket descriptor on success, and -1 on failure.

```
#include <sys/socket.h>

int accept (int sockfd, struct sockaddr *cliaddr, socklen_t *addrlen);
```

Returns: non-negative descriptor if OK, -1 on error

e. **`close()`**: This function is used to close a socket and terminate a connection. It takes a single argument, the socket descriptor, and returns 0 on success and -1 on failure.

```
#include <unistd.h>

int close (int sockfd);
```

Returns: 0 if OK, -1 on error

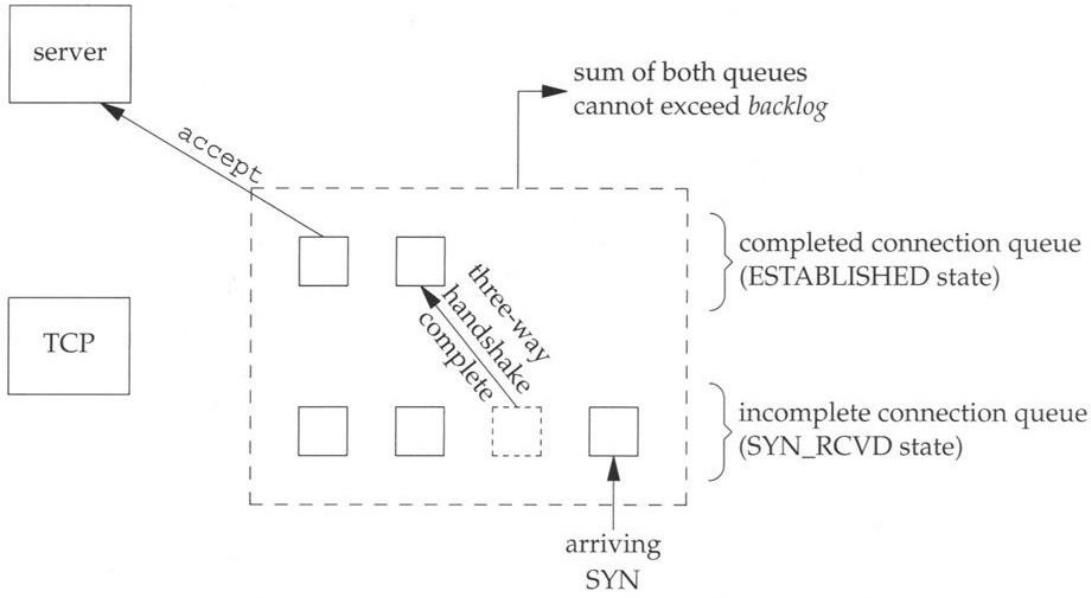
8. With a neat block diagram explain the queues maintained by TCP for a listening socket. Also show the packets exchanged during the connection establishment with these two queues.

Ans:

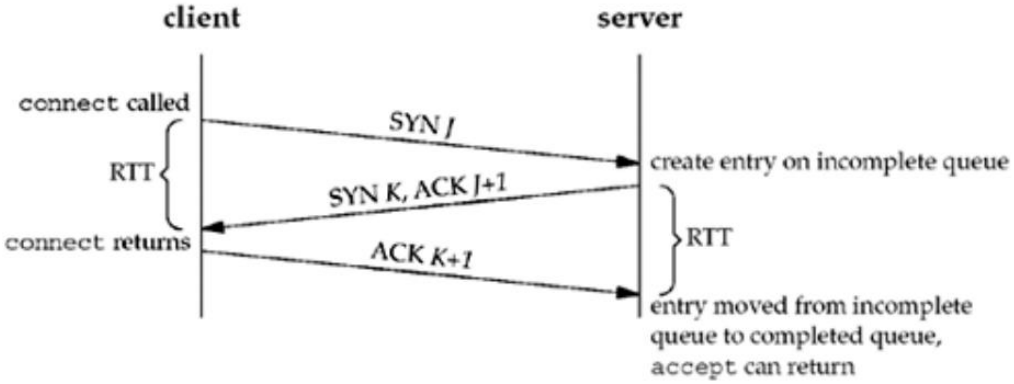
good luck :)

To understand the backlog argument, we must realize that for a given listening socket, the kernel maintains two queues:

1. An **incomplete connection queue**, which contains an entry for each SYN that has arrived from a client for which the server is awaiting completion of the TCP three-way handshake. These sockets are in the SYN_RCVD state.
2. A **completed connection queue**, which contains an entry for each client with whom the TCP three-way handshake has completed. These sockets are in the ESTABLISHED state.



- When an entry is created on the incomplete queue, the parameters from the listen socket are copied over to the newly created connection.
- The connection creation mechanism is completely automatic; the server process is not involved.
- Packets exchanged during the connection establishment with these two queues.



- When a SYN arrives from a client, TCP creates a new entry on the incomplete queue and then responds with the second segment of the three-way handshake: the server's SYN with an ACK of the client's SYN
- This entry will remain on the incomplete queue until the third segment of the threeway handshake arrives (the client's ACK of the server's SYN), or until the entry times out.
- If the three-way handshake completes normally, the entry moves from the incomplete queue to the end of the completed queue.

- When the process calls accept, which we will describe in the next section, the first entry on the completed queue is returned to the process, or if the queue is empty, the process is put to sleep until an entry is placed onto the completed queue.

9. Illustrate the significance of fork and exec functions.

Ans:

a. fork:

```
#include <unistd.h>
```

```
pid_t fork(void);
```

Returns: 0 in child, process ID of child in parent, -1 on error

- It returns once in the calling process (called the parent) with a return value that is the process ID of the newly created process (the child). It also returns once in the child, with a return value of 0. Hence, the return value tells the process whether it is the parent or the child.
- The reason fork returns 0 in the child, instead of the parent's process ID, is because a child has only one parent and it can always obtain the parent's process ID by calling getppid.
- There are two typical uses of fork:
  1. A process makes a copy of itself so that one copy can handle one operation while the other copy does another task. This is typical for network servers. We will see many examples of this later in the text.
  2. A process wants to execute another program. Since the only way to create a new process is by calling fork, the process first calls fork to make a copy of itself, and then one of the copies (typically the child process) calls exec (described next) to replace itself with the new program. This is typical for programs such as shells.

b. exec

- The only way in which an executable program file on disk can be executed by Unix is for an existing process to call one of the six **`exec`** functions.
- **`exec`** replaces the current process image with the new program file, and this new program normally starts at the main function.

good luck :)

```
#include <unistd.h>

int execl (const char *pathname, const char *arg0, ... /* (char *) 0 */ );

int execv (const char *pathname, char *const argv[]);

int execle (const char *pathname, const char *arg0, ...
                /* (char *) 0, char *const envp[] */ );

int execve (const char *pathname, char *const argv[], char *const envp[]);

int execlp (const char *filename, const char *arg0, ... /* (char *) 0 */ );

int execvp (const char *filename, char *const argv[]);
```

All six return: -1 on error, no return on success

10. Outline the typical concurrent server with the help of pseudocode.

Ans:

- The simplest way to write a concurrent server under Unix is to fork a child process to handle each client.

```
pid_t pid;
int listenfd, connfd;
listenfd = Socket( ... );
/* fill in sockaddr_in{} with server's well-known port */
Bind(listenfd, ... );
Listen(listenfd, LISTENQ);
for ( ; ; ) {
    connfd = Accept (listenfd, ... ); /* probably blocks */
    if( (pid = Fork()) == 0) {
        Close(listenfd); /* child closes listening socket */
        doit(connfd); /* process the request */
        Close(connfd); /* done with this client */
        exit(0); /* child terminates */
    }
    Close(connfd); /* parent closes connected socket */
}
```

- When a connection is established, accept returns, the server calls fork, and the child process services the client (on connfd, the connected socket) and the parent process waits for another connection (on listenfd, the listening socket).
- The parent closes the connected socket since the child handles the new client.

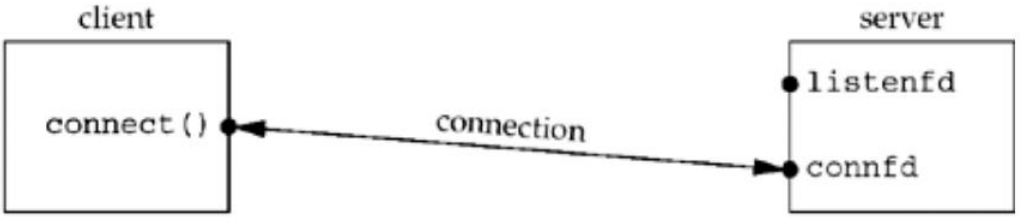
11. Demonstrate the status of client/ server before and after call to *accept* returns with a neat block diagram.

Ans:

Status of client/server before call to accept returns.

- Above figure shows the status of the client and server while the server is blocked in the call to accept and the connection request arrives from the client.

Status of client/server after return from accept

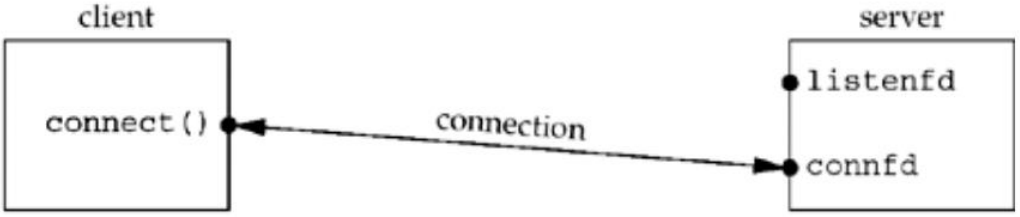


- Immediately after accept returns, we have the scenario shown in Figure. The connection is accepted by the kernel and a new socket, connfd, is created. This is a connected socket and data can now be read and written across the connection.

12. Demonstrate the status of client/ server after fork returns with a neat block diagram.

Ans:

Status of client/server after return from accept

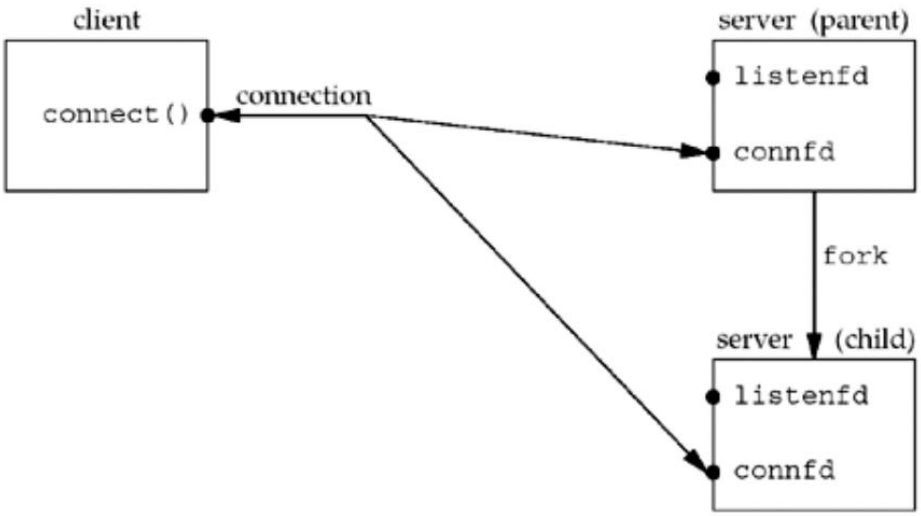


- Immediately after accept returns, we have the scenario shown in Figure. The connection is accepted by the kernel and a new socket, connfd, is created. This is a connected socket and data can now be read and written across the connection.

Status of client/server after fork returns.

- The next step in the concurrent server is to call fork. Figure 4.16 shows the status after fork returns.

good luck :)



- Notice that both descriptors, listenfd and connfd, are shared (duplicated) between the parent and child.

13. Comment on the significance of *getsockname* and *getpeername* functions..

Ans:

- These two functions return either the local protocol address associated with a socket (getsockname) or the foreign protocol address associated with a socket (getpeername).

```
#include <sys/socket.h>

int getsockname(int sockfd, struct sockaddr *localaddr, socklen_t *addrlen);

int getpeername(int sockfd, struct sockaddr *peeraddr, socklen_t *addrlen);

                                        Both return: 0 if OK, -1 on error
```

- Notice that the final argument for both functions is a value-result argument. That is, both functions fill in the socket address structure pointed to by localaddr or peeraddr.

These two functions are required for the following reasons:

- After connect successfully returns in a TCP client that does not call bind, getsockname returns the local IP address and local port number assigned to the connection by the kernel.
- After calling bind with a port number of 0, getsockname returns the local port number that was assigned.
- **`getsockname`** can be called to obtain the address family of a socket.
- In a TCP server that binds the wildcard IP address, once a connection is established with a client, the server can call getsockname to obtain the local IP address assigned to the connection.
- The socket descriptor argument in this call must be that of the connected socket, and not the listening socket.
- When a server is execed by the process that calls accept, the only way the server can obtain the identity of the client is to call getpeername.

14. Develop the pseudocode that returns the address family of a socket.

good luck :)

Ans:

```c
//Address Family of a Socket

#include "unp.h"

int sockfd_to_family(int sockfd) {
    struct sockaddr_storage ss;
    socklen_t len = sizeof(ss);

    Getsockname(sockfd, (SA *)&ss, &len);

    return (ss.ss_family);
}
```

good luck :)

Self-Learning Topics:

15. Develop the 'C' program to demonstrate the TCP echo server: main function

Ans:

```c
//TCP Echo Server : main()

#include "unp.h"

int main(int argc, char **argv) {
    int listenfd = Socket(AF_INET, SOCK_STREAM, 0);

    struct sockaddr_in clientAddress, serverAddress;
    bzero(&serverAddress, sizeof(serverAddress));
    serverAddress.sin_family = AF_INET;
    serverAddress.sin_addr.s_addr = htonl(INADDR_ANY);
    serverAddress.sin_port = htons(SERV_PORT);

    Bind(listenfd, (SA *)&serverAddress, sizeof(serverAddress));

    Listen(listenfd, LISTENQ);

    for (;;) {
        socklen_t clilen = sizeof(clientAddress);
        int client = Accept(listenfd, (SA *)&clientAddress, &clilen);
        str_echo(client);
        Close(client);
    }
}
```

good luck :)

16. Develop the 'C' program to demonstrate the TCP echo client: str_cli function

Ans:

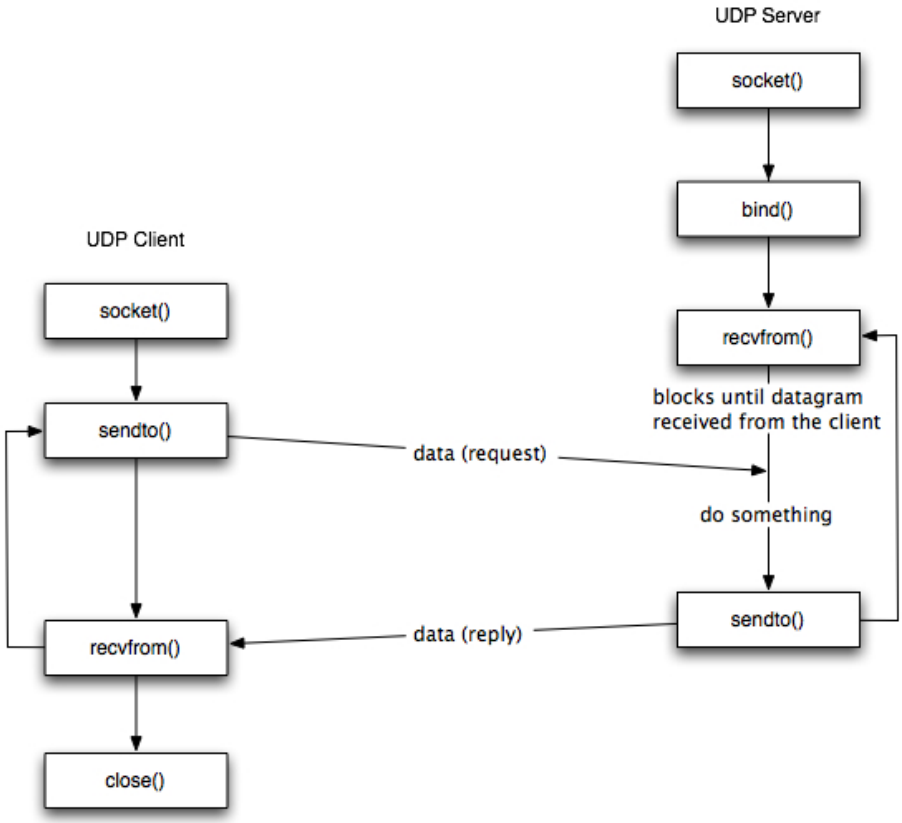
```c
//TCP Echo Client : str_cli()

#include "unp.h"

void str_cli(FILE *fp, int sockfd) {
    char sendline[MAXLINE], recvline[MAXLINE];

    while (Fgets(sendline, MAXLINE, fp) != NULL) {
        Writen(sockfd, sendline, strlen (sendline));
        Readline(sockfd, recvline, MAXLINE);
        Fputs(recvline, stdout);
    }
}
```

good luck :)

Unit 3: (Elementary UDP Sockets)

1. **Illustrate the significance of socket functions for UDP TCP client/server with a neat block diagram.**

Ans:



- The Figure shows the function calls for a typical UDP client/server.
- The client does not establish a connection with the server. Instead, the client just sends a datagram to the server using the **`sendto`** function (described in the next section), which requires the address of the destination (the server) as a parameter.
- Similarly, the server does not accept a connection from a client. Instead, the server just calls the **`recvfrom`** function, which waits until data arrives from some client.
- **`recvfrom`** returns the protocol address of the client, along with the datagram, so the server can send a response to the correct client.
- Figure shows a timeline of the typical scenario that takes place for a UDP client/server exchange.

good luck :)

2. Explain the following functions of UDP socket:

a. recvfrom, b. sendto

Ans:

a. recvfrom:

```
#include <sys/socket.h>

ssize_t recvfrom(int sockfd, void *buff, size_t nbytes, int flags, struct
sockaddr *from, socklen_t *addrlen);
```

- This function is used to receive data from a socket using the User Datagram Protocol (UDP). It takes the following arguments:
    - socket descriptor (int)
    - buffer to store the received data (void*)
    - maximum size of the buffer (size_t)
    - flags (int)
    - address of the sender (struct sockaddr*)
    - size of the sender address (socklen_t*)
- The function reads data from the socket and stores it in the buffer. It also stores the address of the sender in the address struct provided. The flags argument can be used to specify options such as non-blocking mode. The function returns the number of bytes received, or -1 on error.

b. sendto:

```
#include <sys/socket.h>

ssize_t sendto(int sockfd, const void *buff, size_t nbytes, int flags, const
struct sockaddr *to, socklen_t addrlen);
```

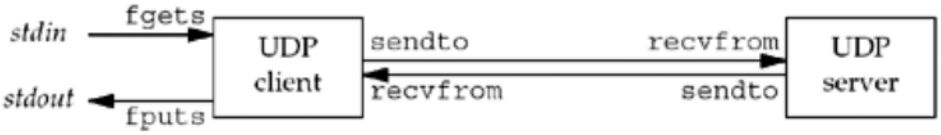
Both return: number of bytes read or written if OK, – 1 on error

- This function is used to send data to a socket using the User Datagram Protocol (UDP). It takes the following arguments:
    - socket descriptor (int)
    - buffer containing the data to be sent (void*)
    - size of the data to be sent (size_t)
    - flags (int)
    - address of the receiver (struct sockaddr*)
    - size of the receiver address (socklen_t)
- The function sends the data in the buffer to the specified address using the socket descriptor. The flags argument can be used to specify options such as non-blocking mode. The function returns the number of bytes sent, or -1 on error.
- Note that UDP is a connectionless protocol, so the client and server do not need to establish a connection before sending or receiving data. Therefore, bind() and connect() functions are not used in UDP sockets.

good luck :)

3. List and explain with a neat block diagram the steps associated with simple UDP echo client and server.

Ans:



1. Server: Create a socket
   - The server creates a socket using the socket() function.
   - The function returns a socket descriptor, which is used to identify the socket in all future operations.
2. Server: Bind the socket to an address and port
   - The server binds the socket to a specific address and port using the bind() function.
   - This step is used to associate the socket with a specific IP address and port, so that it can receive incoming data.
3. Server: Receive data
   - The server uses the recvfrom() function to receive data from the socket.
   - The function stores the received data in the buffer and the address of the sender in the address struct provided.
4. Server: Send data back
   - The server uses the sendto() function to send the data back to the sender.
   - The function sends the data in the buffer to the specified address using the socket descriptor.
5. Server: Close the socket
   - The server closes the socket using the close() function, which takes the socket descriptor as its argument.
6. Client: Create a socket
   - The client creates a socket using the socket() function.
   - The function returns a socket descriptor, which is used to identify the socket in all future operations.
7. Client: Send data
   - The client uses the sendto() function to send data to the server.
   - The function sends the data in the buffer to the specified address using the socket descriptor.
8. Client: Receive data
   - The client uses the recvfrom() function to receive data from the socket.
   - The function stores the received data in the buffer and the address of the sender in the address struct provided.
9. Client: Close the socket
   - The client closes the socket using the close() function, which takes the socket descriptor as its argument.

- This step is used to close the connection between the client and server. This ensures that any resources associated with the socket are freed and the socket is no longer available for communication.

4. Develop the 'C' program to demonstrate the UDP echo server: main function.

Ans:

```c
//UDP Echo Server : main()

#include "unp.h"

int main(int argc, char **argv) {
    int sockfd = Socket(AF_INET, SOCK_DGRAM, 0);

    struct sockaddr_in serverAddress, clientAddress;
    bzero(&serverAddress, sizeof(serverAddress));
    serverAddress.sin_family = AF_INET;
    serverAddress.sin_addr.s_addr = htonl(INADDR_ANY);
    serverAddress.sin_port = htons(SERV_PORT);

    Bind(sockfd, (SA *)&serverAddress, sizeof(serverAddress));

    dg_echo(sockfd, (SA *)&clientAddress, sizeof(clientAddress));
}
```

5. Develop the 'C' program to demonstrate the UDP echo server: dg_echo function

Ans:

```c
//UDP Echo Server : dg_echo()

#include "unp.h"

void dg_echo(int sockfd, SA *pcliaddr, socklen_t clilen) {
    char mesg[MAXLINE];

    for (;;) {
        socklen_t len = clilen;
        int n = Recvfrom(sockfd, mesg, MAXLINE, 0, pcliaddr, &len);
        Sendto(sockfd, mesg, n, 0, pcliaddr, len);
    }
}
```

6. Develop the 'C' program to demonstrate the UDP echo client: main function.

Ans:

```c
// UDP Echo Client : main()

#include "unp.h"

int main(int argc, char **argv) {
    if (argc != 2)
        err_quit("usage: udpcli <IPaddress>");

    int sockfd = Socket(AF_INET, SOCK_DGRAM, 0);

    struct sockaddr_in serverAddress;
    bzero(&serverAddress, sizeof(serverAddress));
    serverAddress.sin_family = AF_INET;
    serverAddress.sin_port = htons(SERV_PORT);
    Inet_pton(AF_INET, argv[1], &serverAddress.sin_addr);

    dg_cli(stdin, sockfd, (SA *)&serverAddress, sizeof(serverAddress));
}
```

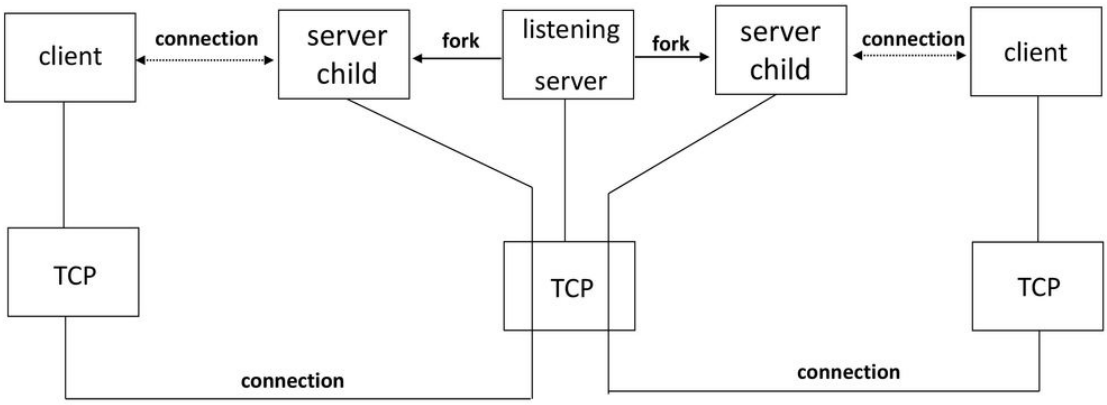
7. Develop the 'C' program to demonstrate the UDP echo client: dg_cli function.

Ans:

```c
// UDP Echo Client : dg_cli()
#include "unp.h"

void dg_cli(FILE *fp, int sockfd, const SA *serverAddress, socklen_t servlen) {
    char sendline[MAXLINE], recvline[MAXLINE + 1];

    while (Fgets(sendline, MAXLINE, fp) != NULL) {
        Sendto(sockfd, sendline, strlen(sendline), 0, serverAddress, servlen);
        int n = Recvfrom(sockfd, recvline, MAXLINE, 0, NULL, NULL);
        recvline[n] = '\0';
        printf("%s\n", recvline);
    }
}
```

8. Outline the summary of TCP client/server with two clients.

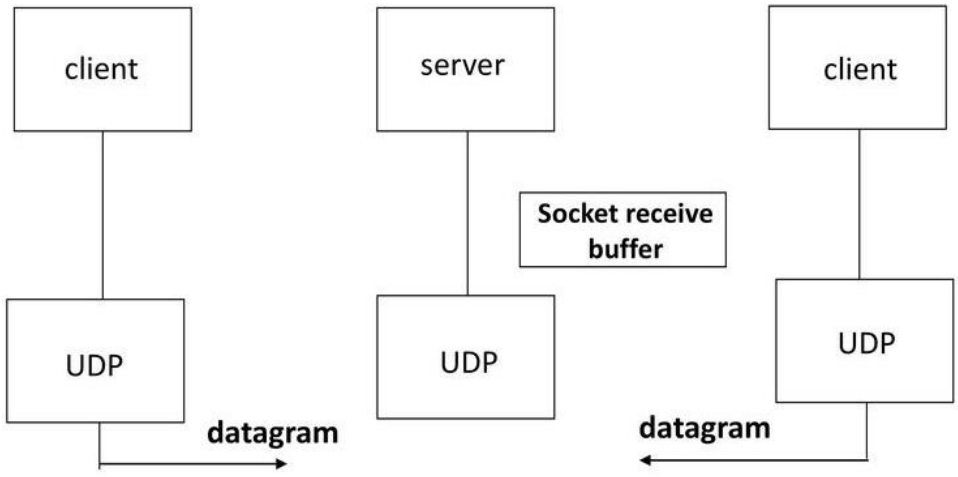
Ans:



1. Server: Creates a socket.
2. Server: Binds the socket to an address and port.
3. Server: Listens for incoming connections.
4. Server: Accepts incoming connections.
5. Client 1: Creates a socket.
6. Client 1: Connects to the server.
7. Client 1: Sends and receives data.
8. Client 2: Creates a socket.
9. Client 2: Connects to the server.
10. Client 2: Sends and receives data.
11. Server: Closes the socket.
12. Client 1: Closes the socket.
13. Client 2: Closes the socket.

9. Outline the summary of UDP client/server with two clients.

Ans:



1. Server: Creates a socket.
2. Server: Binds the socket to an address and port.

3. Client 1: Creates a socket.
4. Client 1: Sends data to the server using sendto function.
5. Server: Receives data from client 1 using recvfrom function.
6. Server: Sends a reply to client 1 using sendto function.
7. Client 2: Creates a socket.
8. Client 2: Sends data to the server using sendto function.
9. Server: Receives data from client 2 using recvfrom function.
10. Server: Sends a reply to client 2 using sendto function.
11. Server: Closes the socket.
12. Client 1: Closes the socket.
13. Client 2: Closes the socket.

10. Develop the 'C' program for dg_cli function that verifies returned socket address.

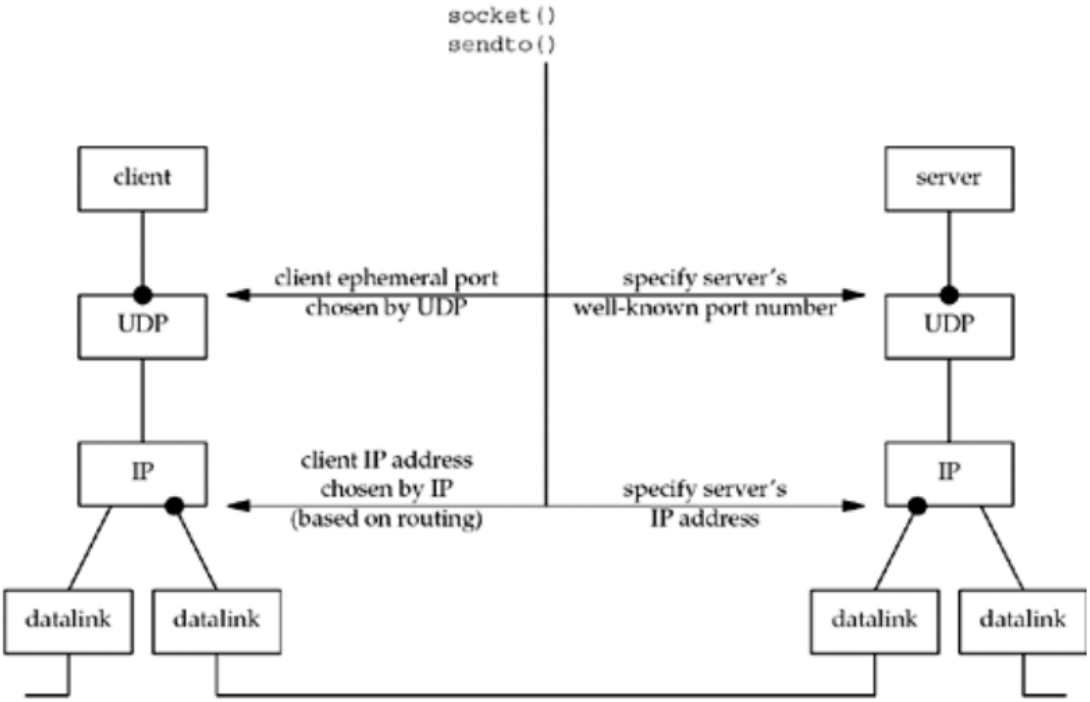
Ans:

```c
/* Verify Returned Socket Address */

#include "unp.h"

void dg_cli(FILE *fp, int sockfd, const SA *pservaddr, socklen_t servlen) {
    char sendline[MAXLINE], recvline[MAXLINE + 1];

    struct sockaddr *preply_addr;
    preply_addr = Malloc(servlen);

    while (Fgets(sendline, MAXLINE, fp) != NULL) {
        Sendto(sockfd, sendline, strlen(sendline), 0, pservaddr, servlen);
        socklen_t len = servlen;
        int n = Recvfrom(sockfd, recvline, MAXLINE, 0, preply_addr, &len);

        if (len != servlen || memcmp(pservaddr, preply_addr, len) != 0) {
            printf("reply from %s (ignored)\n", Sock_ntop(preply_addr, len));
            continue;
        }

        recvline[n] = '\0';
        printf("%s\n", recvline);
    }
}
```

11. Outline the summary of UDP client/server from client's perspective with a neat. block diagram.

Ans:

good luck :)

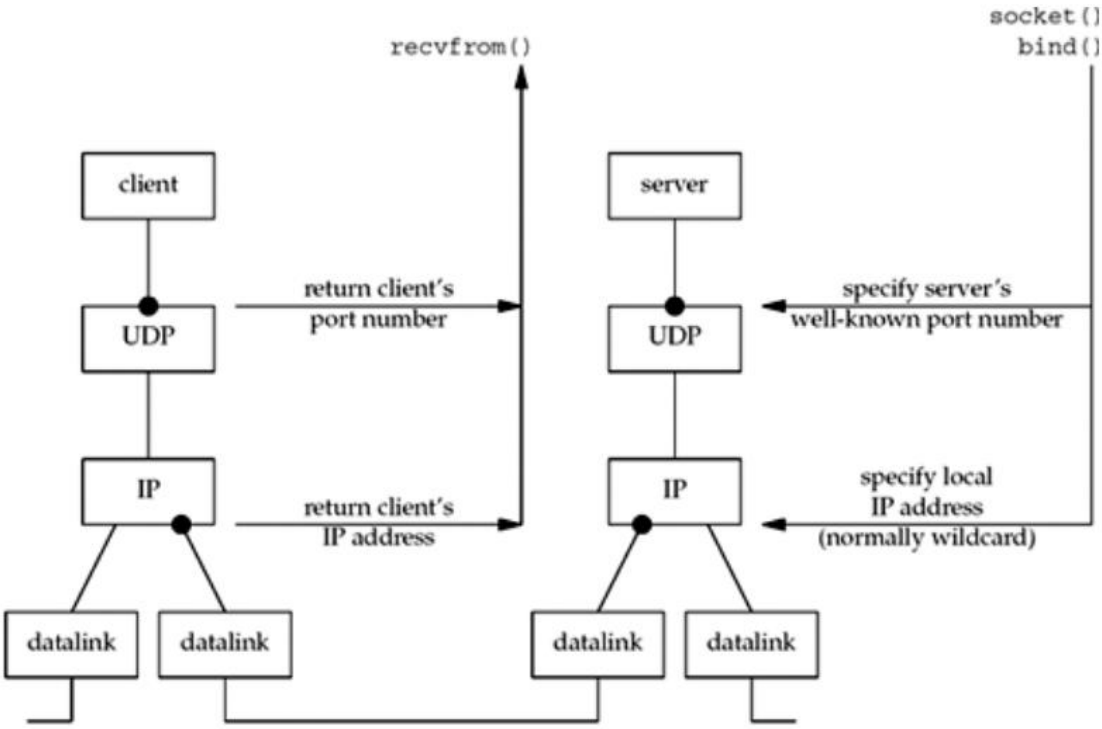


- The client must specify the server's IP address and port number for the call to sendto. Normally, the client's IP address and port are chosen automatically by the kernel, although we mentioned that the client can call bind if it so chooses.
- If these two values for the client are chosen by the kernel, we also mentioned that the client's ephemeral port is chosen once, on the first sendto, and then it never changes.
- The client's IP address, however, can change for every UDP datagram that the client sends, assuming the client does not bind a specific IP address to the socket.
- The reason is shown in Figure. If the client host is multihomed, the client could alternate between two destinations, one going out the datalink on the left, and the other going out the datalink on the right.
- What happens if the client binds an IP address to its socket, but the kernel decides that an outgoing datagram must be sent out some other datalink? In this case the IP datagram will contain a source IP address that is different from the IP address of the outgoing datalink

12. Outline the summary of UDP client/server from server's perspective with a neat block diagram.

Ans:

good luck :)



- There are at least four pieces of information that a server might want to know from an arriving IP datagram: the source IP address, destination IP address, source port number, and destination port number.
- Figure shows the function calls that return this information for a TCP server and a UDP server.

13. Develop the 'C' program to demonstrate the UDP dg_cli function that calls connect.
Ans:

good luck :)

```c
#include "unp.h"
void dg_cli(FILE *fp, int sockfd, const SA *pservaddr, socklen_t servlen){
    int n;
    char sendline[MAXLINE], recvline[MAXLINE + 1];
    Connect(sockfd, (SA *) pservaddr, servlen);
    while (Fgets(sendline, MAXLINE, fp) != NULL) {
        Write(sockfd, sendline, strlen(sendline));
        n = Read(sockfd, recvline, MAXLINE);
        recvline[n] = 0; /* null terminate */
        Fputs(recvline, stdout);
    }
}
```

14. Develop the 'C' program to demonstrate the UDP dg_cli function that writes a fixed number of datagrams to the server.

Ans:

```c
//To write a fixed number of datagrams to the server

#include "unp.h"
#define NDG 2000
#define DGLEN 1400

void dg_cli(FILE *fp, int sockfd, const SA *pservaddr, socklen_t servlen) {
    char sendline[DGLEN];
    for (int i = 0; i < NDG; i++) {
        Sendto(sockfd, sendline, DGLEN, 0, pservaddr, servlen);
    }
}
```

Self-Learning Topics:

15. Explain the simple SCTP streaming echo client and server with a neat block diagram.

good luck :)

Ans:

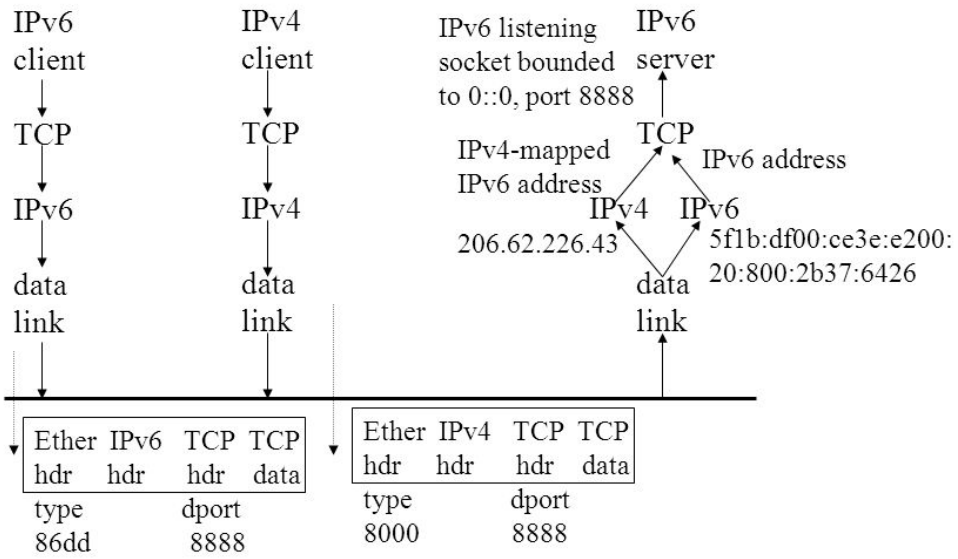
- A client reads a line of text from standard input and sends the line to the server. The line follows the form **`[#] text`**, where the number in brackets is the SCTP stream number on which the text message should be sent.
- The server receives the text message from the network, increases the stream number on which the message arrived by one, and sends the text message back to the client on this new stream number.
- The client reads the echoed line and prints it on its standard output, displaying the stream number, stream sequence number, and text string.
- We show two arrows between the client and server depicting two unidirectional streams being used, even though the overall association is full-duplex. The **`fgets`** and **`fputs`** functions are from the standard I/O library.
- We use the **`sctp_sendmsg`** and **`sctp_recvmsg`** functions to send and receive messages.

Unit 4: (Advanced Sockets-I)

1. **With a neat block diagram explain IPv6 server on dual stack host serving IPv4 and IPv6 clients.**

Ans:

good luck :)



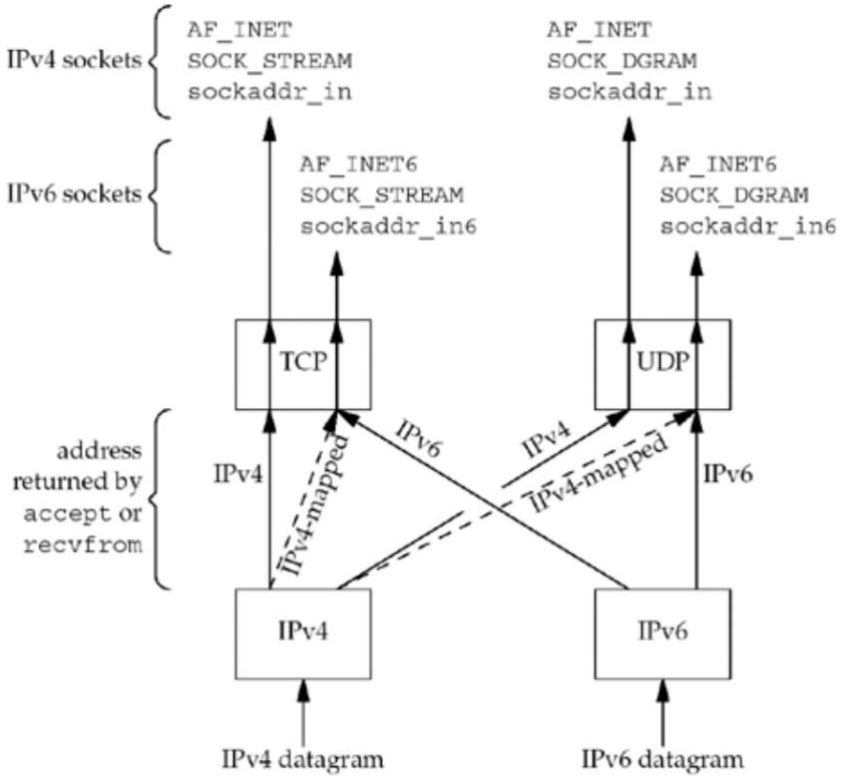
Steps that allow an IPv4 TCP client to communicate with an IPv6 server as follows:

- The IPv6 server starts creating an IPv6 listening socket, and we assume it binds the wildcard address to the socket.
- The IPv4 client calls **`gethostbyname`** and finds an A record for the server. The server host will have both an A record and an AAAA record since it supports both protocols, but the IPv4 client asks for only an A record.
- The client calls connect and the client's host sends an IPv4 SYN to the server.
- The server host receives the IPv4 SYN directed to the IPv6 listening socket, sets a flag indicating that this connection is using IPv4-mapped IPv6 addresses, and responds with an IPv4 SYN/ACK. When the connection is established, the address returned to the server by accept is the IPv4-mapped IPv6 address.
- When the server host sends to the IPv4-mapped IPv6 address, its IP stack generates IPv4 datagrams to the IPv4 address. Therefore, all communication between this client and server takes place using IPv4 datagrams.
- Unless the server explicitly checks whether this IPv6 address is an IPv4-mapped IPv6 address, the server never knows that it is communicating with an IPv4

2. **Explain with a neat block diagram how the received IPv4 and IPv6 datagrams are processed depending on the type of receiving socket.**

Ans:

good luck :)

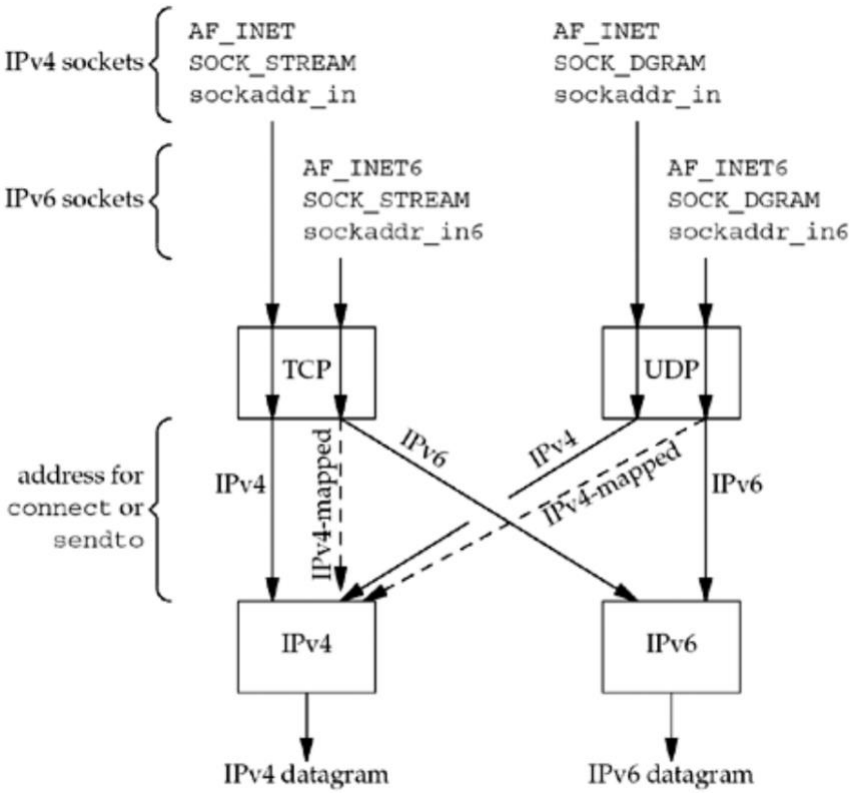


- If an IPv4 datagram is received for an IPv4 socket, nothing special is done. These are the two arrows labelled "IPv4" in the figure: one to TCP and one to UDP. IPv4 datagrams are exchanged between the client and server.
- If an IPv6 datagram is received for an IPv6 socket, nothing special is done. These are the two arrows labelled "IPv6" in the figure: one to TCP and one to UDP. IPv6 datagrams are exchanged between the client and server.
- When an IPv4 datagram is received for an IPv6 socket, the kernel returns the corresponding IPv4-mapped IPv6 address as the address returned by **`accept`** (TCP) or **`recvfrom`** (UDP).
- The converse of the previous bullet is false: In general, an IPv6 address cannot be represented as an IPv4 address; therefore, there are no arrows from the IPv6 protocol box to the two IPv4 sockets.

3. **Explain with a neat block diagram how the client requests are processed depending on the address type and socket type.**

Ans:

good luck :)



- An IPv4 server starts on an IPv4-only host and creates an IPv4 listening socket. The IPv6 client calls **`connect`** with the IPv4-mapped IPv6 address in the IPv6 socket address structure.
- The kernel detects the mapped address and automatically sends an IPv4 SYN to the server.
- The server responds with an IPv4 SYN/ACK, and the connection is established using IPv4 datagrams.
- If an IPv4 TCP client calls **`connect`** specifying an IPv4 address, or if an IPv4 UDP client calls **`sendto`** specifying an IPv4 address, nothing special is done.
- If an IPv6 TCP client calls **`connect`** specifying an IPv6 address, or if an IPv6 UDP client calls **`sendto`** specifying an IPv6 address, nothing special is done.

4. List and explain the numerous ways to start a daemon.

Ans:

good luck :)

1. **Forking the Parent Process:** The parent process forks a child process and exits, allowing the child process to continue running as a daemon. This method is commonly used in Unix-based systems.
2. **Double Forking:** The parent process forks a child process, and the child process forks another child process and exits. This method is also commonly used in Unix-based systems to ensure that the daemon process is not a child of any other process and cannot receive signals from the terminal.
3. **Using 'nohup':** The 'nohup' command can be used to run a program and ignore any hangup signals. This method is commonly used to run a program in the background and continue running even after the terminal is closed.
4. **Detaching the Process:** The process can be detached from the terminal and run in the background by redirecting the input and output to /dev/null. This method is commonly used in Unix-based systems.
5. **Using 'systemd':** Systemd is a service manager that can be used to run a program as a daemon on Linux systems. It provides an easy way to configure, start, stop, and restart daemons.

5. List and explain the actions on startup for syslogd Daemon.

Ans:

1. The configuration file, normally **`/etc/syslog.conf`**, is read, specifying what to do with each type of log message that the daemon can receive. These messages can be appended to a file (a special case of which is the file **`/dev/console`**, which writes the message to the console), written to a specific user (if that user is logged in), or forwarded to the **`syslogd`** daemon on another host.
2. A Unix domain socket is created and bound to the pathname **`/var/run/log`** (**`/dev/log`** on some systems).
3. A UDP socket is created and bound to port 514 (the **`syslog`** service).
4. The pathname **`/dev/klog`** is opened. Any error messages from within the kernel appear as input on this device.

The **`syslogd`** daemon runs in an infinite loop that calls **`select`**, waiting for any one of its three descriptors (from Steps 2, 3, and 4) to be readable; it reads the log message and does what the configuration file says to do with that message. If the daemon receives the **`SIGHUP`** signal, it rereads its configuration file.

6. With a function prototype explain the syslog Function.

Ans:

good luck :)

The syslog function is a system call that is used to generate log messages and send them to the system's log daemon. The function prototype for the syslog function is as follows:

```
#include <syslog.h>

void syslog(int priority, const char *message, ... );
```

- **`priority`:** an integer value representing the priority level of the log message, which can be one of the constants defined in the **`<syslog.h>`** header file.
- **`message/format`:** a pointer to a null-terminated string that specifies the format of the log message, following the same rules as the printf function.
- **`. . .`**: a variable number of arguments, depending on the format string, that will be used to substitute placeholders in the format string.

The syslog function takes the priority and message passed as arguments, adds information like timestamp, process name, host name & sends it to syslogd daemon.

Ex: **`syslog(LOG_ERR. "Error: Failed to open file %s", filename);`**

1. *level* of log messages.

level	Value	Description
LOG_EMERG	0	System is unusable (highest priority)
LOG_ALERT	1	Action must be taken immediately
LOG_CRIT	2	Critical conditions
LOG_ERR	3	Error conditions
LOG_WARNING	4	Warning conditions
LOG_NOTICE	5	Normal but significant condition (default)
LOG_INFO	6	Informational
LOG_DEBUG	7	Debug-level messages (lowest priority)

***facility* of log messages.**

facility	Description
LOG_AUTH	Security/authorization messages
LOG_AUTHPRIV	Security/authorization messages (private)
LOG_CRON	`cron` daemon
LOG_DAEMON	System daemons
LOG_FTP	FTP daemon
LOG_KERN	Kernel messages
LOG_LOCAL0	Local use
LOG_LOCAL1	Local use
LOG_LOCAL2	Local use
LOG_LOCAL3	Local use
LOG_LOCAL4	Local use
LOG_LOCAL5	Local use
LOG_LOCAL6	Local use
LOG_LOCAL7	Local use
LOG_LPR	Line printer system
LOG_MAIL	Mail system
LOG_NEWS	Network news system
LOG_SYSLOG	Messages generated internally by `syslogd`
LOG_USER	Random user-level messages (default)
LOG_UUCP	UUCP system

good luck :)

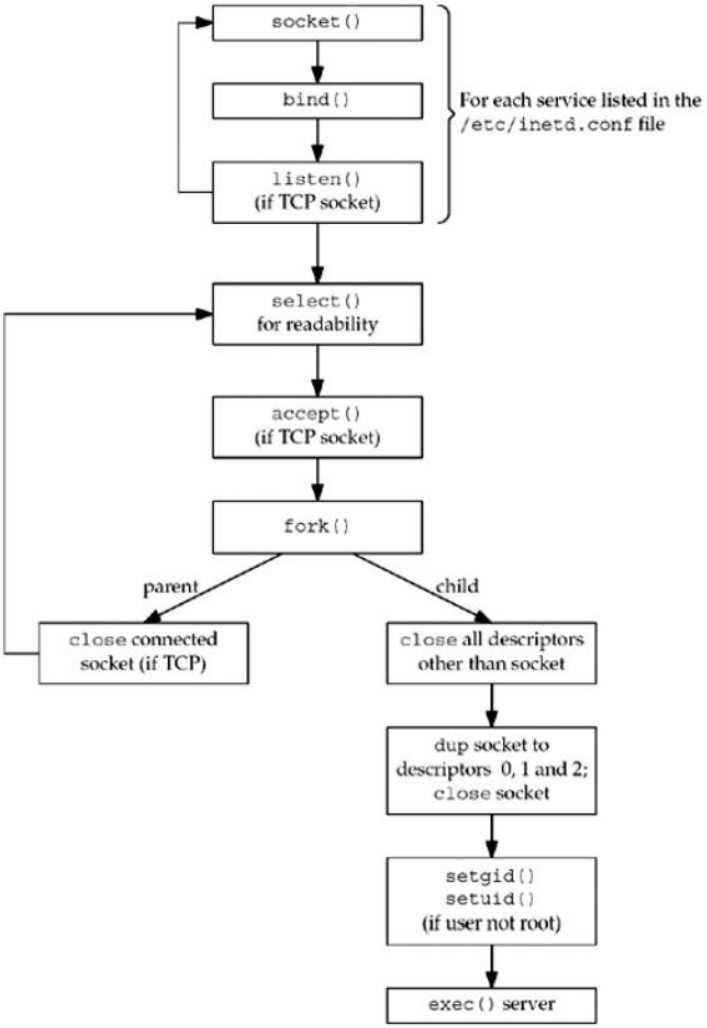
Self- Learning Topics:

7. Explain the significance of daemon_init function.

Ans:

8. List and explain the steps performed by of inetd Daemon.

Ans:

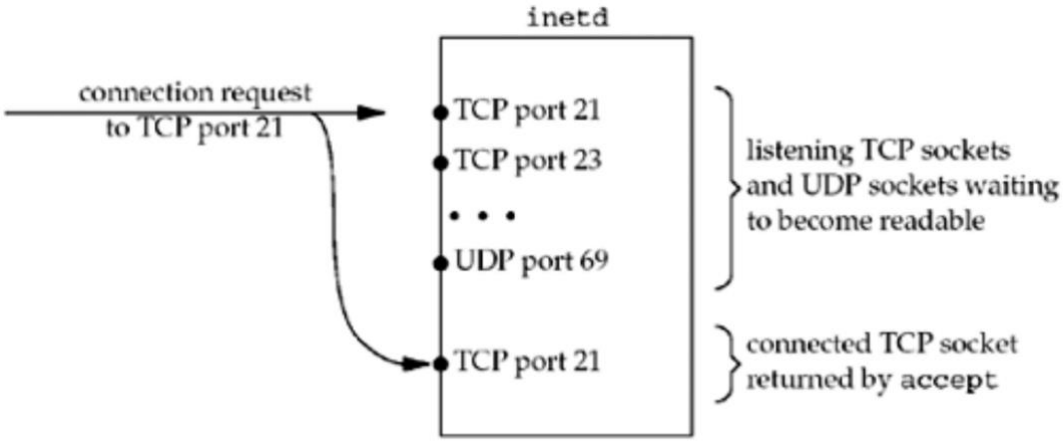


1. On startup, it reads the **`/etc/inetd.conf`** file and creates a socket of the appropriate type (stream or datagram) for all the services specified in the file.
2. **`bind`** is called for the socket, specifying the port for the server and the wildcard IP address. This TCP or UDP port number is obtained by calling **`getservbyname`** with the service-name and protocol fields from the configuration file as arguments.
3. For TCP sockets, **`listen`** is called so that incoming connection requests are accepted. This step is not done for datagram sockets.
4. After all the sockets are created, **`select`** is called to wait for any of the sockets to become readable.

5. When **`select`** returns that a socket is readable, if the socket is a TCP socket and the nowait flag is given, accept is called to accept the new connection
6. The **`inetd`** daemon **`forks`** and the child process handles the service request. The child process now does an **`exec`** to execute the appropriate server-program to handle the request, passing the arguments specified in the configuration file.
7. If the socket is a stream socket, the parent process must close the connected socket (like our standard concurrent server). The parent calls **`select`** again, waiting for the next socket to become readable.

9. With a neat block diagram explain the inetd descriptors when connection request arrives for TCP port.

Ans:



a. The inetd daemon receives the connection request and checks its configuration file to see if it should handle the request for the specified port.
b. If the inetd daemon is configured to handle the request, it forks a new child process to handle the connection.
c. The child process inherits the network socket descriptor from the inetd daemon and uses it to establish a connection with the client.
d. The child process reads the client's request and processes it accordingly. This may involve reading data from the client, writing data to the client, or both.
e. After processing the request, the child process closes the socket descriptor and exits.
f. The inetd daemon continues to monitor the specified port for new connection requests.
g. This process is repeated for each new connection request that arrives for the specified port.
h. inetd daemon can handle multiple ports at the same time and reduce the number of daemons that need to run on the system.

good luck :)

10. Explain the following with a neat block diagram:

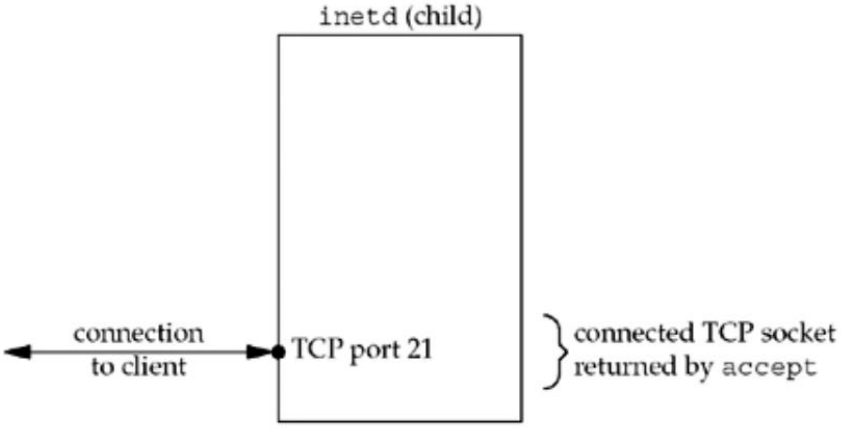
a. inetd descriptors in child

b. inetd descriptors after dup2

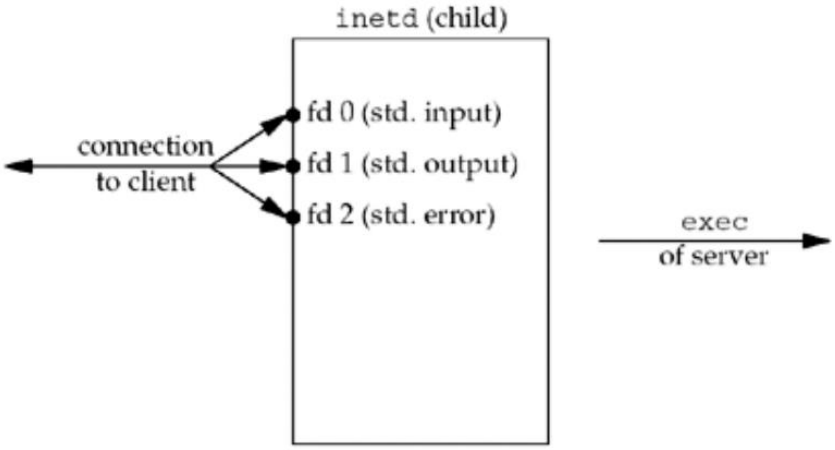
Ans:

//Explaination: gg.

a.



b.



good luck :)

11. Develop the pseudocode for daemon_inetd function to daemonize process run by inetd.

Ans:

```
// daemon_inetd function: daemonizes process run by inetd

#include "unp.h"
#include <syslog.h>

extern int daemon_proc; /* defined in error.c */

void daemon_inetd(const char *pname, int facility) {
    daemon_proc = 1; /* for our err_XXX() functions */
    openlog(pname, LOG_PID, facility);
}
```

Unit 5: (Advanced Sockets-II)

1. **Outline the important points with respect to unicasting, multicasting and broadcasting.**

Ans:

1. Unicasting:
    - Sends data to a specific individual or device.
    - Most commonly used method of communication on the internet.
    - Low network congestion as data is sent only to intended recipient.
    - Can be used for point-to-point communication.
    - Can be used with IP version 4 and IP version 6

2. Broadcasting:
    - Sends data to all devices on a network.
    - Useful for network management and discovery.
    - Can cause network congestion if used excessively.
    - Not typically used for point-to-point communication.
    - Only available on LANs, not WANs or the internet.

3. Multicasting:
    - Sends data to a specific group of recipients on a network.
    - Useful for applications such as streaming video and audio.
    - Can reduce network congestion compared to broadcasting.
    - Can be used with IP version 4 and IP version 6.
    - Can be used for point-to-multipoint communication.

2. List and explain the routing protocols which makes use of multicasting.

Ans:

There are several routing protocols that make use of multicasting, including:

- **PIM (Protocol Independent Multicast)**: PIM is a multicast routing protocol that can work with different unicast routing protocols, such as OSPF or BGP.
- **MOSPF (Multicast Open Shortest Path First)**: MOSPF is a multicast extension of the OSPF routing protocol, which uses multicast groups to send routing updates.
- **DVMRP (Distance Vector Multicast Routing Protocol)**: DVMRP is a multicast routing protocol that uses distance-vector algorithms to find multicast routes.
- **CBT (Core-Based Trees)**: CBT is a multicast routing protocol that uses a shared distribution tree to send multicast traffic.
- **SSM (Source-Specific Multicast)**: SSM is a multicast routing protocol that allows receivers to join multicast groups for specific sources.
- **IGMP (Internet Group Management Protocol)**: IGMP is a protocol used by IP hosts to report their multicast group memberships to any neighboring multicast routers.

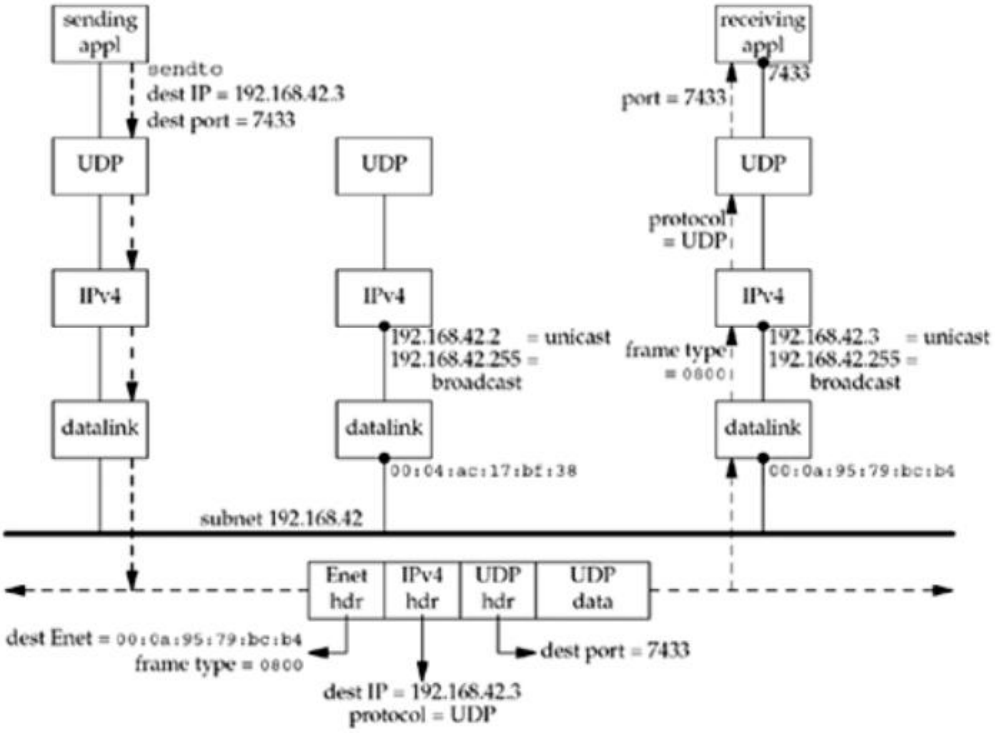
3. Differentiate between unicast and broadcast.

Ans: Refer Q1.

4. Make use of UDP datagram to understand unicasting.

Ans: Explanation: 20.3.

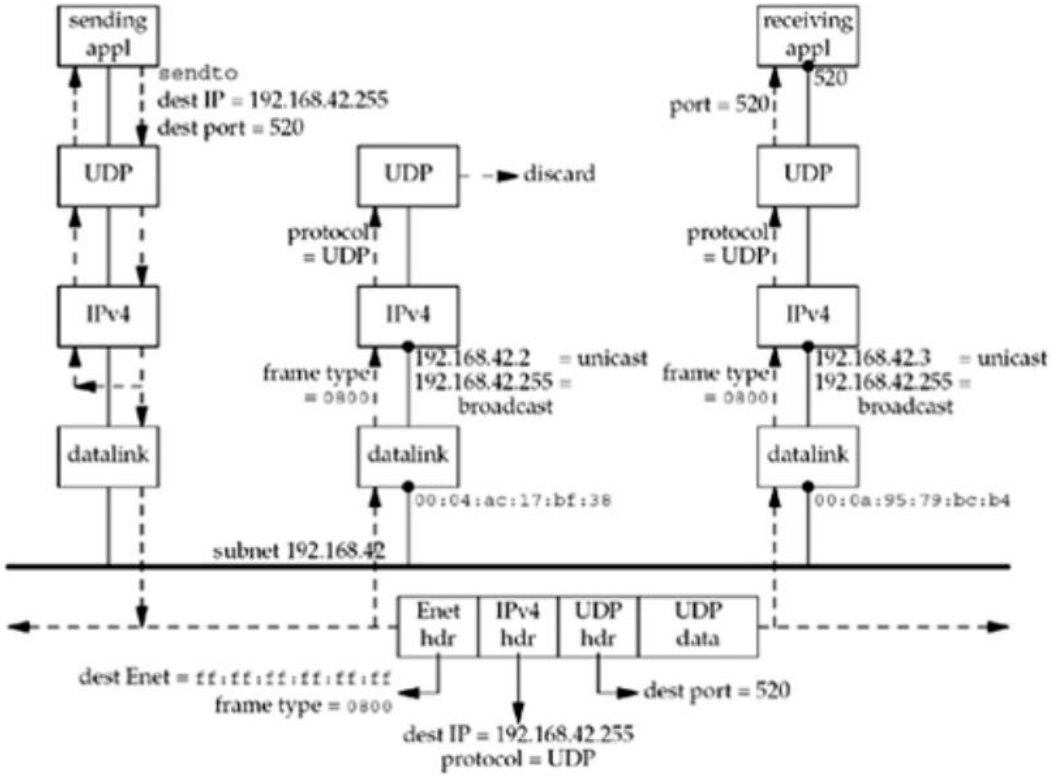
good luck :)



5. Make use of UDP datagram to understand broadcasting.

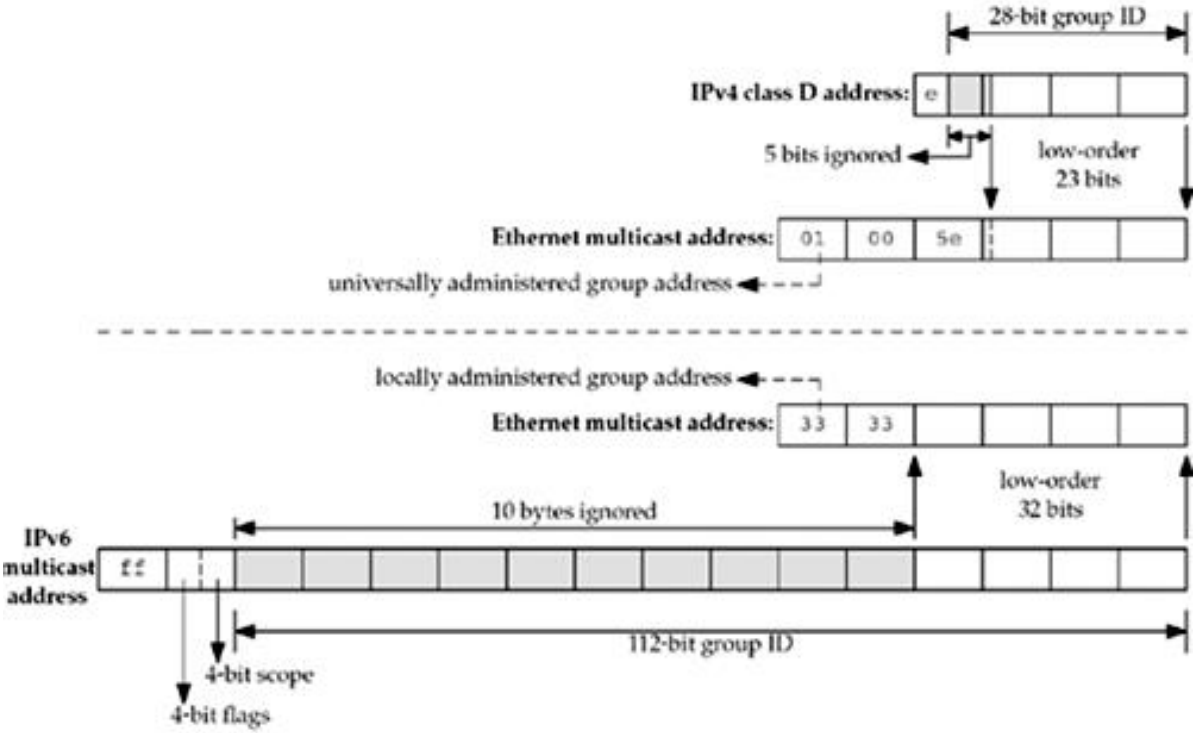
Ans: Explanation: 20.4.

good luck :)



6. Explain with a neat block diagram how IPv4 and IPv6 multicast addresses are mapped to Ethernet addresses.

Ans:



IPv4 and IPv6 multicast addresses are mapped to Ethernet addresses using a process called multicast address resolution protocol (MAR) mapping. The mapping process is slightly different for IPv4 and IPv6.

good luck :)

- For IPv4:
  - The least significant 23 bits of the IPv4 multicast address are used to form the low-order 23 bits of the Ethernet address.
  - The most significant bit of the Ethernet address is set to 1.

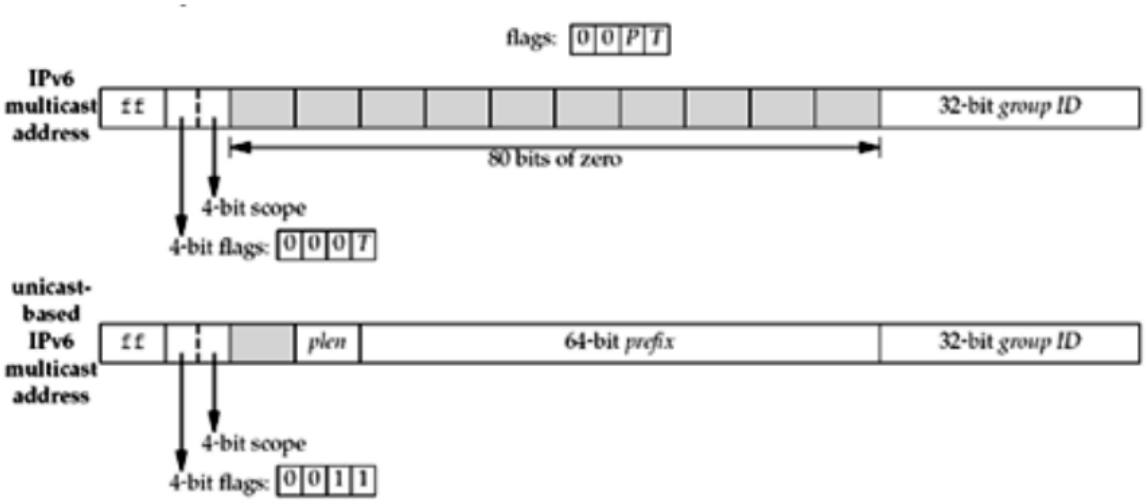
- For IPv6:

  - The low-order 24 bits of the IPv6 multicast address are used to form the low-order 24 bits of the Ethernet address.
  - The next bit is set to 1.
  - The most significant bit of the Ethernet address is set to 1.
  - The resulting Ethernet address is called the multicast MAC address, which is used to identify the specific multicast group on the local network segment.

- For example, if the IPv4 multicast address is 224.0.0.1, the Ethernet multicast address would be 01:00:5E:00:00:01. And if the IPv6 multicast address is FF02:0:0:0:0:0:0:1, the Ethernet multicast address would be 33:33:00:00:00:01.
- It's worth noting that this mapping process is done by the networking stack of the operating system, and it is transparent to the applications.

7. What is the format of IPv6 multicast addresses?

Ans: 21.2



The format of an IPv6 multicast address is as follows:

- The first 8 bits (most significant bits) of an IPv6 multicast address are always set to 11111111 (binary) or FF (hexadecimal).
- The next 4 bits are the flags field and are set to 0010.
- The next 4 bits are the scope field and can have the following values:
  - 0001: Interface-local scope (nodes on the same interface)
  - 0010: Link-local scope (nodes on the same link or subnet)
  - 0100: Site-local scope (nodes in the same site)

    - 1000: Organization-local scope (nodes in the same organization)
    - 1111: Global scope (nodes on the entire internet)
- The remaining 112 bits make up the group ID field and are used to identify the specific multicast group.

So the IPv6 multicast address format is represented as
"FFxx:xxxx:xxxx:xxxx:xxxx:xxxx:xxxx:xxxx" where x's are the group ID field.

For example, the multicast address FF02:0:0:0:0:0:0:1 is a link-local scope multicast address that identifies all nodes on the same link or subnet.

8. Illustrate the scope of multicast addresses.

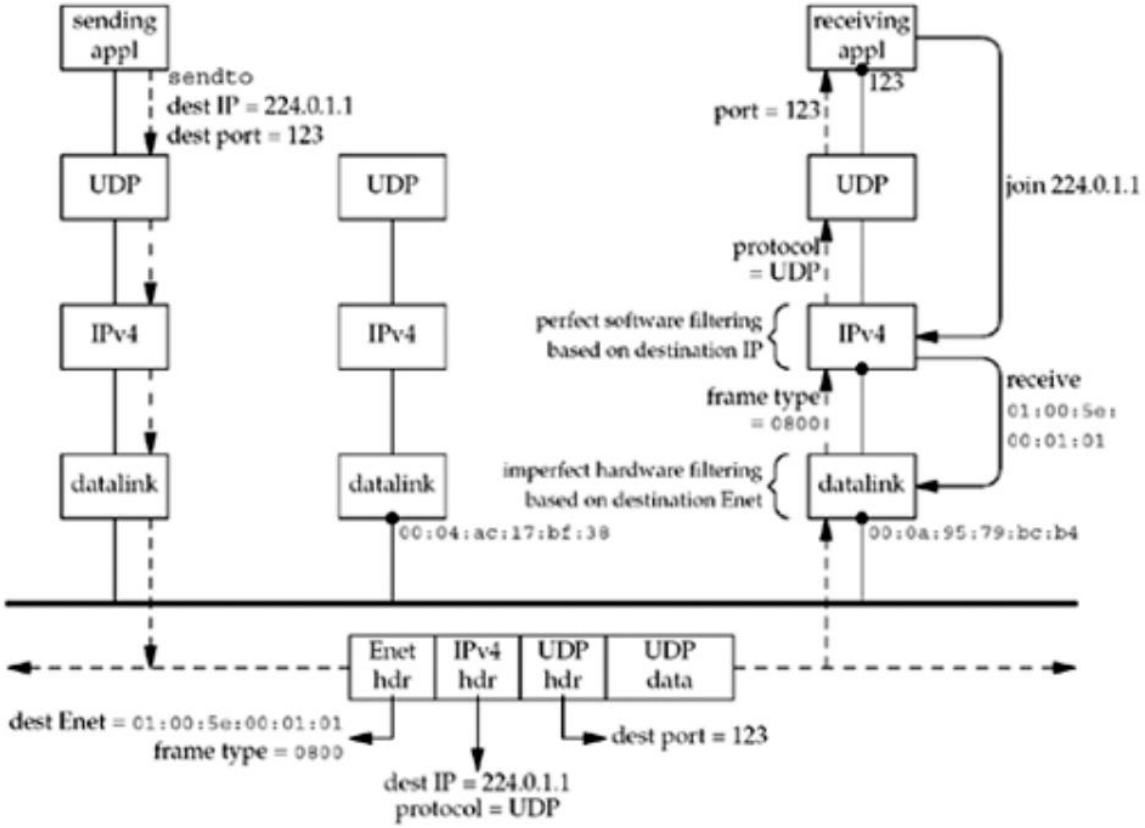
Ans:
Multicast addresses have a scope, which refers to the range of networks over which the multicast traffic can be forwarded. The scope of a multicast address is determined by the value of the four most significant bits of the address.

- Interface-local scope: Addresses in this range, from 224.0.0.0 to 224.0.0.255, are only valid on the local network interface. They are not forwarded by routers.
- Link-local scope: Addresses in this range, from 224.0.1.0 to 224.0.1.255, are only valid on the local network segment. They are not forwarded beyond the local network segment.
- Admin-local scope: Addresses in this range, from 224.0.2.0 to 238.255.255.255, are intended for local administration and are not forwarded by routers.
- Site-local scope: Addresses in this range, from 239.0.0.0 to 239.255.255.255, are only valid within a single site. They are not forwarded beyond the site boundary.
- Global scope: Addresses in this range, from 240.0.0.0 to 255.255.255.255, are valid across the entire internet. They are forwarded by routers.

9. Illustrate with a neat block diagram, multicast example of a UDP datagram.

Ans: 21.4

good luck :)



10. Illustrate the NTP packet format and definitions with the help of ntp.h header.

Ans:

good luck :)

```c
#define JAN_1970 2208988800UL /* 1970 - 1900 in seconds */

struct l_fixedpt { /* 64-bit fixed-point */
  uint32_t int_part;
  uint32_t fraction;
};

struct s_fixedpt { /* 32-bit fixed-point */
  uint16_t int_part;
  uint16_t fraction;
};

struct ntpdata { /* NTP header */
  u_char status;
  u_char stratum;
  u_char ppoll;
  int precision:8;
  struct s_fixedpt distance;
  struct s_fixedpt dispersion;
  uint32_t refid;
  struct l_fixedpt reftime;
  struct l_fixedpt org;
  struct l_fixedpt rec;
  struct l_fixedpt xmt;
};

#define VERSION_MASK 0x38
#define MODE_MASK 0x07
#define MODE_CLIENT 3
#define MODE_SERVER 4
#define MODE_BROADCAST 5
```

Only for understanding purposes, refer to the information below. (NOT TO BE WRITTEN IN EXAM)

- **#define JAN_1970 2208988800UL** - This line defines a macro named JAN_1970 with the value 2208988800UL. It represents the number of seconds between January 1st, 1900, and January 1st, 1970.

good luck :)

- **struct l_fixedpt** - This line starts the definition of a structure named l_fixedpt, which is a 64-bit fixed-point number. It consists of two 32-bit integer variables int_part and fraction.
- **struct s_fixedpt** - This line starts the definition of a structure named s_fixedpt, which is a 32-bit fixed-point number. It consists of two 16-bit integer variables int_part and fraction.
- **struct ntpdata** - This line starts the definition of a structure named ntpdata, which is used to store the header of an NTP packet. It contains the following fields:
    - **u_char status** - an 8-bit unsigned integer to store the status of the NTP packet.
    - **u_char stratum** - an 8-bit unsigned integer to store the stratum of the NTP packet.
    - **u_char ppoll** - an 8-bit unsigned integer to store the poll interval of the NTP Packet.
    - **int precision:8** - an 8-bit integer to store the precision of the NTP Packet.
    - **struct s_fixedpt distance** - a s_fixedpt structure to store the distance of the NTP Packet.
    - **struct s_fixedpt dispersion** - a s_fixedpt structure to store the dispersion of the NTP Packet.
    - **uint32_t refid** - a 32-bit unsigned integer to store the reference identifier of the NTP Packet.
    - **struct l_fixedpt reftime** - a l_fixedpt structure to store the reference time of the NTP Packet.
    - **struct l_fixedpt org** - a l_fixedpt structure to store the origin time of the NTP Packet.
    - **struct l_fixedpt rec** - a l_fixedpt structure to store the receive time of the NTP Packet.
    - **struct l_fixedpt xmt** - a l_fixedpt structure to store the transmit time of the NTP Packet.
- **#define VERSION_MASK 0x38** - This line defines a macro named VERSION_MASK with the value 0x38. It represents the version number of NTP.
- **#define MODE_MASK 0x07** - This line defines a macro named MODE_MASK with the value 0x07. It represents the mode of NTP.
- **#define MODE_CLIENT 3** - This line defines a macro named MODE_CLIENT with the value 3. It represents the client mode of NTP.
- **#define MODE_SERVER 4** - This line defines a macro named MODE_SERVER with the value 4. It represents the server mode of NTP.
- **#define MODE_BROADCAST 5** - This line defines a macro named MODE_BROADCAST with the value 5. It represents the broadcast mode of NTP.